|  |
| --- |
| URCA – UFR SCIENCES |
| RAPPORT INFO0401 |
| Concepts d’Algorithmie avancée |

|  |
| --- |
| Luca ALVARO, Antoine THEBAULT  01/01/2020 |

# Sommaire

[Sommaire 1](#_Toc36900843)

[Algorithmes de base 4](#_Toc36900844)

[Normalisation de données d’un tableau (Dimension 1) 4](#_Toc36900845)

[Calcul du maximum de 2 images binaires 6](#_Toc36900846)

[Multiplication de 2 matrices carrées 9](#_Toc36900847)

[Calcul d’un nombre parfait 12](#_Toc36900848)

[Concaténer 2 tableaux 13](#_Toc36900849)

[Suite de Fibonacci avec un Vecteur 16](#_Toc36900850)

[Fibonacci avec des Variables 17](#_Toc36900851)

[Majorité d’un tableau 18](#_Toc36900852)

[Conversion d’un nombre base 10 en nombre binaire 20](#_Toc36900853)

[Algorithmes de Tri 22](#_Toc36900854)

[Tri via un Arbre 22](#_Toc36900855)

[Tri de l’arbre 22](#_Toc36900856)

[Tri à bulle 25](#_Toc36900857)

[Tri par Sélection 26](#_Toc36900858)

[Algorithmes de probabilité 28](#_Toc36900859)

[Simuler un jeu de tennis 28](#_Toc36900860)

[Algorithmes de formalisme de pointeur 30](#_Toc36900861)

[Copier une chaîne de caractère 30](#_Toc36900862)

[Supprimer l'occurrence 31](#_Toc36900863)

[Inverser les éléments d’un tableau 34](#_Toc36900864)

[Palindrome 36](#_Toc36900865)

[Copier un tableau 37](#_Toc36900866)

[Algorithmes récursifs 39](#_Toc36900867)

[Ackerman récursif 39](#_Toc36900868)

[Tri d’un tableau par pivot (tri rapide) récursif 40](#_Toc36900869)

[Algorithmes de Pile 42](#_Toc36900870)

[Définition des structures 42](#_Toc36900871)

[Créer une Pile 42](#_Toc36900872)

[Empiler une Pile 43](#_Toc36900873)

[Dépiler une Pile 44](#_Toc36900874)

[Sommet d’une pile 44](#_Toc36900875)

[Déterminer si une pile est vide 45](#_Toc36900876)

[Vider une Pile 45](#_Toc36900877)

[Afficher une Pile 46](#_Toc36900878)

[Egalité de deux Piles 47](#_Toc36900879)

[Trier une Pile 48](#_Toc36900880)

[Ackerman avec Pile (Iteratif) 50](#_Toc36900881)

[Algorithmes de Liste 53](#_Toc36900882)

[Définition des structures 53](#_Toc36900883)

[Créer une liste 53](#_Toc36900884)

[La liste est-elle vide ? 54](#_Toc36900885)

[Afficher la liste 54](#_Toc36900886)

[Chercher un élément dans la liste 55](#_Toc36900887)

[Ajouter en dernière position 56](#_Toc36900888)

[Ajouter la valeur à la nième position 58](#_Toc36900889)

[Retirer la valeur à la dernière position 60](#_Toc36900890)

[Retirer la valeur à la position n 62](#_Toc36900891)

[Retirer la valeur (première occurrence) 63](#_Toc36900892)

[Afficher Liste 64](#_Toc36900893)

[La liste est vide ? 65](#_Toc36900894)

[Vider la liste 66](#_Toc36900895)

[Egalité de deux listes 66](#_Toc36900896)

[Construire une liste binaire à partir d’un tableau 67](#_Toc36900897)

[Incrémentation de 1 une liste binaire 68](#_Toc36900898)

[Trier une liste 70](#_Toc36900899)

[Fusionner deux listes triées 71](#_Toc36900900)

[Extraire une chaine de la liste 72](#_Toc36900901)

[Algorithmes d’Arbre 74](#_Toc36900902)

[Définition des structures 74](#_Toc36900903)

[Créer un Arbre 74](#_Toc36900904)

[L’arbre est-il vide ? 75](#_Toc36900905)

[Choix du parcours pour l’affichage 75](#_Toc36900906)

[Parcours Prefixé 76](#_Toc36900907)

[Parcours Infiné 76](#_Toc36900908)

[Parcours Suffixe 77](#_Toc36900909)

[Calcul de la hauteur 77](#_Toc36900910)

[Parcours pour le calcul de la hauteur 78](#_Toc36900911)

[L’arbre est-il équilibré ? 78](#_Toc36900912)

[Deux Arbres sont-ils égaux ? 79](#_Toc36900913)

[Parcours pour l’égalité 79](#_Toc36900914)

[Parcours en largeur 80](#_Toc36900915)

[Vida(n)ge de l’Arbre 83](#_Toc36900916)

[Vidage (récursif) 83](#_Toc36900917)

[Ajout Logique 84](#_Toc36900918)

[Ajout en Largeur 85](#_Toc36900919)

[Equilibrage de l’arbre 88](#_Toc36900920)

[Algorithmes de Huffman 92](#_Toc36900921)

[Définition des structures 92](#_Toc36900922)

[Recherche du maximum 92](#_Toc36900923)

[Transformation de la table de priorité 93](#_Toc36900924)

[Décodage de Huffman 95](#_Toc36900925)

[Encodage de Huffman 97](#_Toc36900926)

[Algorithmes d’Arbre R&N 100](#_Toc36900927)

[Vérifier si un arbre R&N respecte les règles 100](#_Toc36900928)

**Ci-joint :**

* L’ensemble des codes des Algorithmes en C
* Un lanceur avec exemples
* Deux versions de ce dossier (PDF/DOCX)

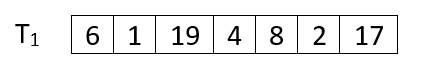
Tous les algorithmes sont accessibles via le lien suivant (GitHub du projet)

***https://github.com/Angom8/INFO0401***

# Algorithmes de base

## Normalisation de données d’un tableau (Dimension 1)

Soit un tableau d’entier T1, on désire exploiter ses données pour une autre réalisation, pour cela nous aurons besoin de normaliser les données de ce tableau sur l’intervalle [a, b].



Comme la plus grande valeur c’est 19 dans T1 alors on doit diviser l’ensemble des valeurs par 19.

**Algorithme :** Normalisation

**Données :** \* t1 : tableau d’entiers;

\* t2 : tableau de réels;

i, n, a, b : entiers;

max : réel;

**Début**

// Lecture de la taille

lire(n);

// Lecture de l’intervalle [a, b]

lire(a);

lire(b);

// Permute a et b si besoins

**Si** a > b **Alors**

a ← a + b;

b ← a – b;

a ← a – b;

**FinSi**

// Lecture de t1

t1 ← allouer(taille(entier)\*n);

**Pour** i allant de 1 à n **Faire**

lire( t1(i) );

**FinPour**

// Affichage de t1

**Pour** i allant de 1 à n **Faire**

afficher(t1(i));

**FinPour**

// Recherche de la valeur maximale

max ← t1(1);

**Pour** i allant de 1 à n **Faire**

**Si** t1[i] > max **Alors**

max ← t1(i);

**FinSi**

**FinPour**

afficher(max);

t2 ← allouer(taille(réel)\*n);

// Normalisation

**Pour** i allant de 1 à n **Faire**

**Si** a < 0 **Alors**

t2[i] ← (t1[i] div  max)\*(b + absolue(a)) + a;

**Sinon**

t2[i] ← (t1[i] div  max)\*(b – a) + a;

**FinSi**

afficher(t2[i]);

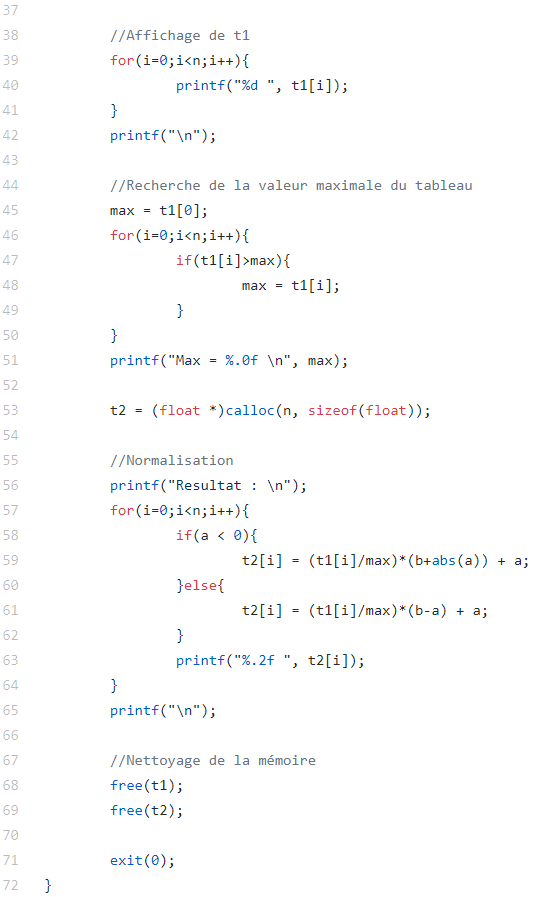
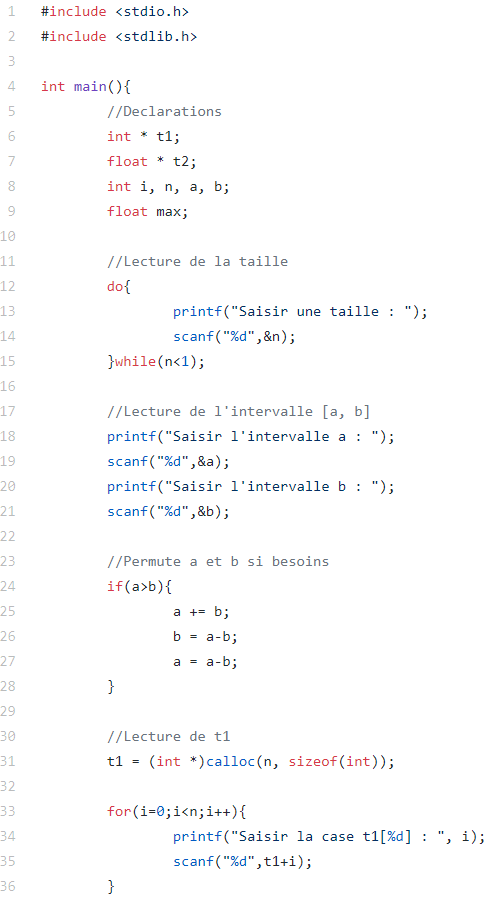
**FinPour**

// Nettoyage de la mémoire

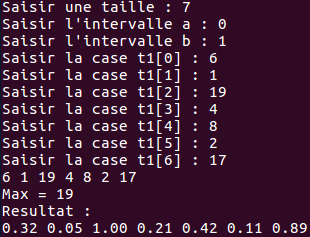
liberer(t1);

liberer(t2);

**Fin**



**Exemple normalisation entre [0, 1] :**



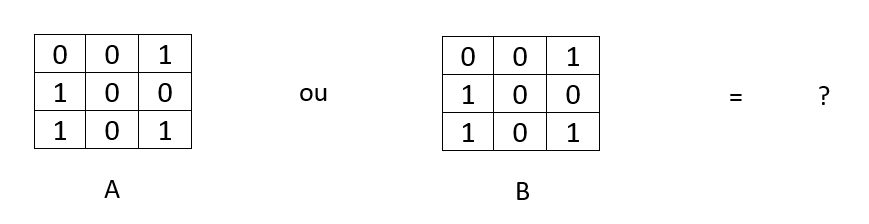
**Complexité :**

Affichage = 2n+1

Affectation = Si a>b : n+6 Sinon : n+3

Addition = Si a>b : n+1 Sinon : n

## Calcul du maximum de 2 images binaires



**Algorithme :** CalculMaxMatricesBinaires

**Données :** \*\* MA, \*\* MB, \*\* MC : matrice d’entiers;

j, i, n : entiers;

**Début**

// Lecture de la taille

lire(n);

// Allocation dynamique

MA ← allouer(taille(\* entier)\*n);

MB ← allouer(taille(\* entier)\*n);

MC ← allouer(taille(\* entier)\*n);

**Pour** i allant de 1 à n **Faire**

MA(i) ← allouer(taille(entier)\*n);

MB(i) ← allouer(taille(entier)\*n);

MC(i) ← allouer(taille(entier)\*n);

**FinPour**

// Lecture de la matrice A

**Pour** i allant de 1 à n **Faire**

**Pour** j allant de 1 à n **Faire**

lire(MA(i, j));

**FinPour**

**FinPour**

// Lecture de la matrice B

**Pour** i allant de 1 à n **Faire**

**Pour** j allant de 1 à n **Faire**

lire(MB(i, j));

**FinPour**

**FinPour**

// Affichage de la matrice A

**Pour** i allant de 1 à n **Faire**

**Pour** j allant de 1 à n **Faire**

afficher(MA(i, j));

**FinPour**

**FinPour**

// Affichage de la matrice B

**Pour** i allant de 1 à n **Faire**

**Pour** j allant de 1 à n **Faire**

afficher(MB(i, j));

**FinPour**

**FinPour**

// Calcul du max de 2 images binaires

**Pour** i allant de 1 à n **Faire**

**Pour** j allant de 1 à n **Faire**

**Si** MA(i, j) > MB(i, j) **Alors**

MC(i, j) ← MA(i, j);

**Sinon**

MC(i, j) ← MB(i, j);

**FinSi**

**FinPour**

**FinPour**

// Affichage de la matrice C, le résultat

**Pour** i allant de 1 à n **Faire**

**Pour** j allant de 1 à n **Faire**

afficher(MC(i, j));

**FinPour**

**FinPour**

// Nettoyage de la mémoire

**Pour** j allant de 1 à n **Faire**

liberer(MA(i));

liberer(MB(i));

liberer(MC(i));

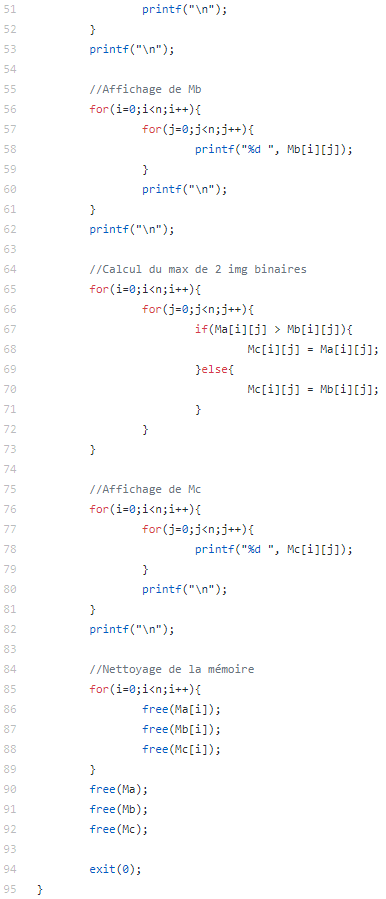
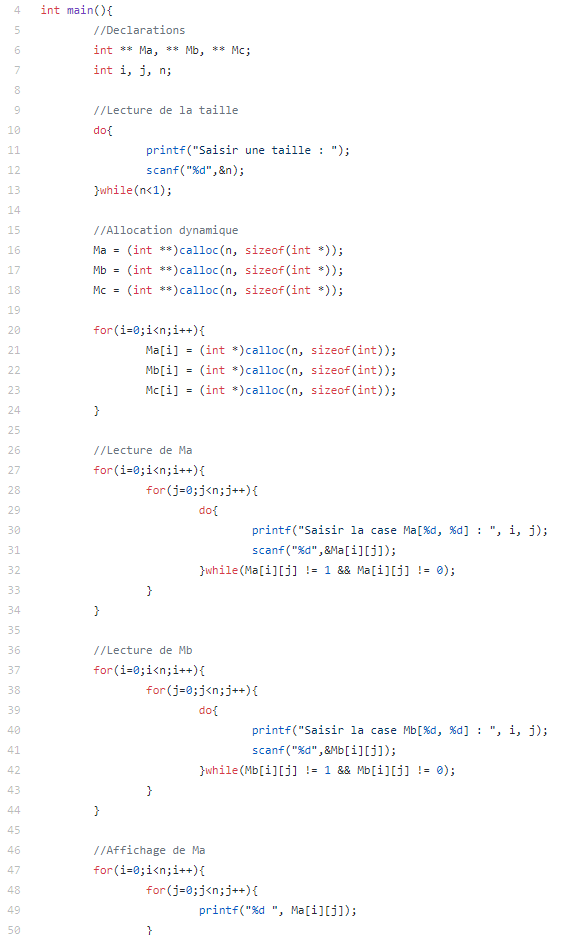
**FinPour**

liberer(MA);

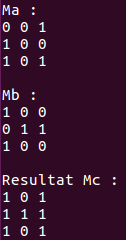
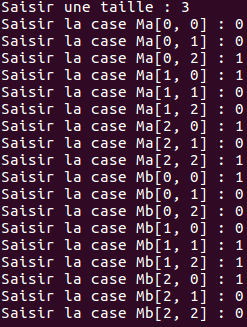
liberer(MB);

liberer(MC);

**Fin**



**Exemple de calcul du max de 2 matrices binaires :**



**Complexité :**

Affichage = 3n2

Affectation = n2+3n+3

Lecture = 2n2+1

## Multiplication de 2 matrices carrées

**Algorithme :** ProduitMatriciel

**Données :** \*\* Ma, \*\* Mb, \*\* Mc : matrice d’entiers;

i, j, k, n : entiers;

**Début**

// Lecture de la taille

lire(n) ;

// Allocation dynamique

MA ← allouer(taille(\* entier)\*n);

MB ← allouer(taille(\* entier)\*n);

MC ← allouer(taille(\* entier)\*n);

**Pour** i allant de 1 à n **Faire**

MA(i) ← allouer(taille(entier)\*n);

MB(i) ← allouer(taille(entier)\*n);

MC(i) ← allouer(taille(entier)\*n);

**FinPour**

// Lecture de la matrice A. On admet que lire() gère la lecture de matrices comme fait précédemment

lire(MA) ;

// Lecture de la matrice B

lire(MB) ;

// Affichage de la matrice A. On admet que afficher() gère l’affichage de matrices comme fait précédemment

afficher(MA) ;

// Affichage de la matrice B

afficher(MB) ;

// Multiplication de la matrice A et B. On fait la somme des a(i, k)\*b(k, j)

**Pour** i allant de 1 à n **Faire**

**Pour** j allant de 1 à n **Faire**

MC(i, j) ← 0 ;

**Pour** k allant de 1 à n **Faire**

MC(i, j) ← MC(i, j) + (MA(i,k)\*MB(k, j)) ;

**FinPour**

**FinPour**

**FinPour**

// Affichage de la matrice C, le résultat

afficher(MC) ;

// Nettoyage de la mémoire

**Pour** j allant de 1 à n **Faire**

liberer(MA(i));

liberer(MB(i));

liberer(MC(i));

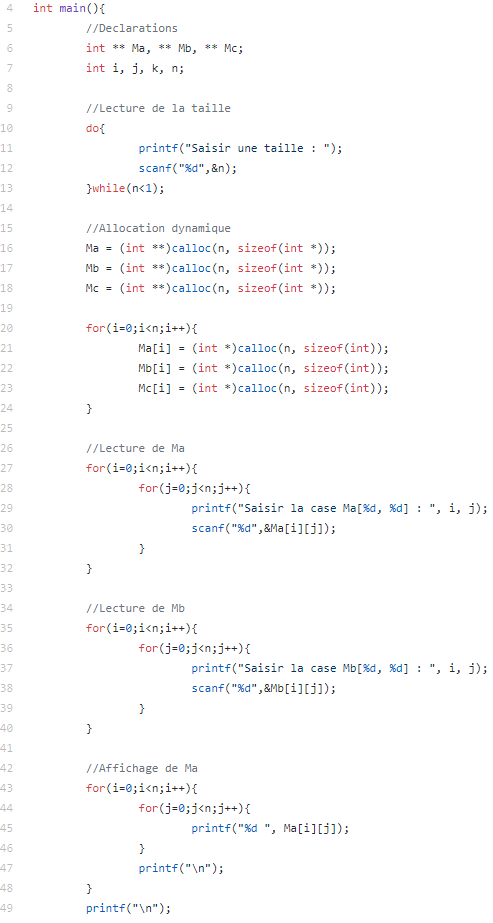
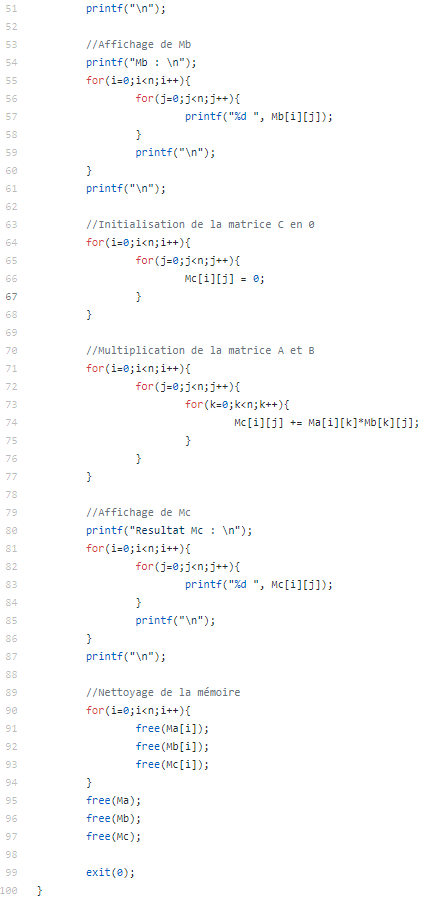
**FinPour**

liberer(MA);

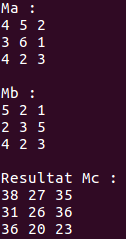
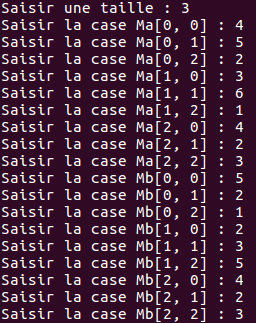
liberer(MB);

liberer(MC);

**Fin**

**Exemple de multiplication de 2 matrices :**



**Complexité :**

Affichage = 3n2

Affectation = n3+n2+3n+3

Lecture = 2n2+1

## Calcul d’un nombre parfait

**Algorithme :** estParfait

**Données :** i, nb, tmp : entiers

**Début**

// Lecture du nombre

lire(nb) ;

// Initialisation

tmp←0 ;

// Calcul d’un nombre parfait

**Pour** i allant de 1 à nb div 2 **Faire**

**Si** nb%i = 0 **Alors**

tmp ← tmp + i ;

**FinSi**

**FinPour**

// Vérifier si le nombre est parfait

**Si** nb = tmp **Alors**

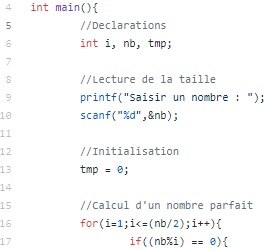
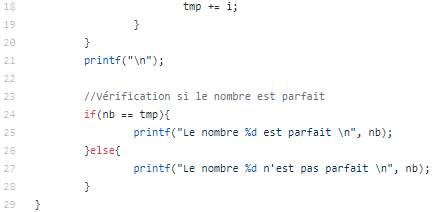
afficher(“Le nombre n est parfait“) ;

**Sinon**

afficher(“Le nombre n n’est pas parfait“) ;

**FinSi**

**Fin**

**Exemple de calcul de nombre parfait :**



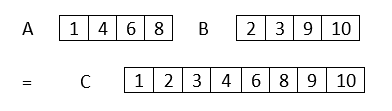
**Complexité :**

Affichage = 1

Affectation = k+1 (k le nb de fois ou nb%i == 0)

Lecture = 1

## Concaténer 2 tableaux



Les Tableaux sont déjà triés et la concaténation doit respecter l’ordre des tableaux d’origine.

**Algorithme :** ConcatenationTableau

**Données :** i, j, n1, n2 : entiers

\* a, \* b, \* c : tableaux d’entiers

**Début**

// Lecture de la taille

lire(n1) ;

lire(n2) ;

// Allocation dynamique

a ← allouer(taille(\* entier)\*n1);

b ← allouer(taille(\* entier)\*n2);

c ← allouer(taille(\* entier)\*(n1+n2));

// Lecture du tableau A

**Pour** i allant de 1 à n1 **Faire**

Lire(a(i));

**FinPour**

// Lecture du tableau B

**Pour** i allant de 1 à n2 **Faire**

Lire(b(i));

**FinPour**

// Affichage du tableau A

afficher(a) ;

// Affichage du tableau B

afficher(b) ;

// Concatener A et B

i ← 0;

j ← 0;

**TantQue** i < n1 **OU** j < n2 **Faire**

**Si** i >= n1 **Alors**

c(i+j) ← b(j);

j ← j+1;

**Sinon Si** j >= n2 **Alors**

c(i+j) ← a(i);

i ← i+1;

**Sinon Si** a(i) < b(i) **Alors**

c(i+j) ← a(i);

i ← i+1;

**Sinon**

c(i+j) ← b(j);

j ← j+1;

**Finsi**

**FinSi**

**FinSi**

**FinTantQue**

// Affichage du tableau C

afficher(c) ;

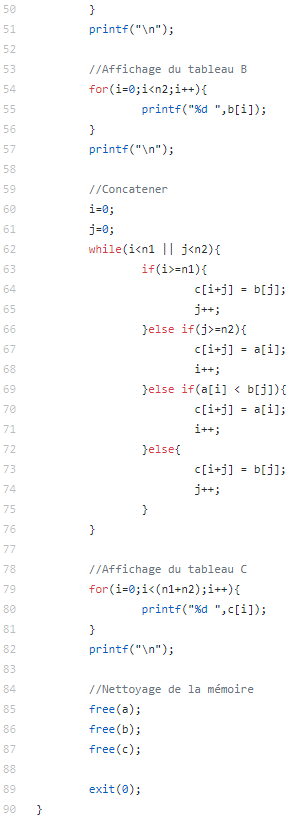
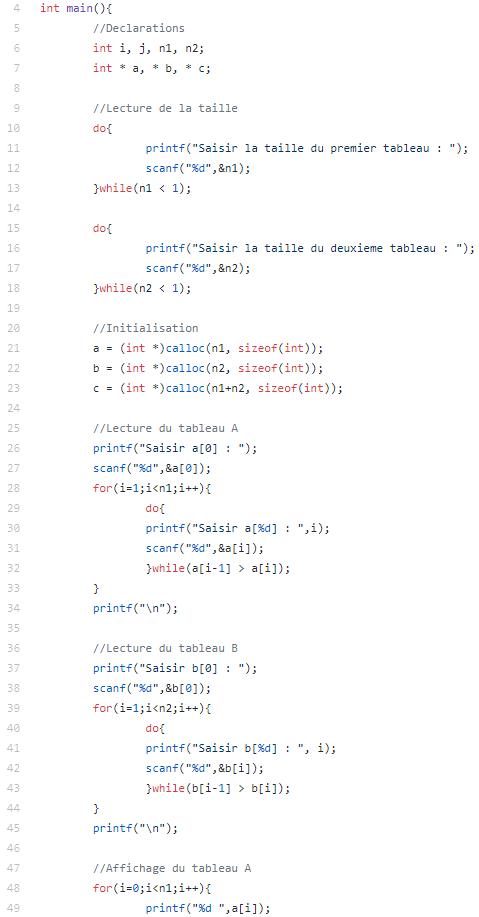
// Nettoyage de la mémoire

liberer(a);

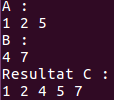
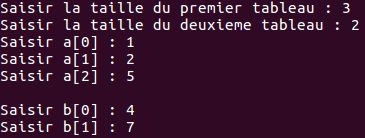
liberer(b);

liberer(c);

**Fin**



**Exemple concatenation de 2 tableaux :**



**Complexité :**

Affichage = 2(n1+n2)

Affectation = 2(n1+n2) + 2

Lecture = n1+n2+2

## Suite de Fibonacci avec un Vecteur

**Algorithme :** VecteurFibonacci

**Données :** i, n : entiers;

\* tab : tableau d’entiers;

**Début**

// Lecture du nombre d’opération

lire(n);

// Initialisation

tab ← allouer(taille(entier)\*n);

tab(0) ← 0;

tab(1) ← 1;

// Calcul de la suite de Fibonacci

**Pour** i allant de 2 à n **Faire**

tab(i) ← tab(i-1) + tab(i-2);

**FinPour**

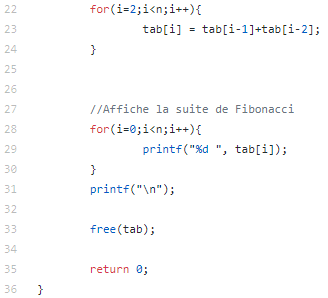
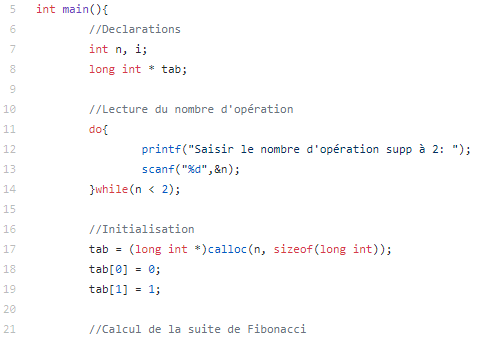
// Afficher la suite de Fibonacci

**Pour** i allant de 1 à n **Faire**

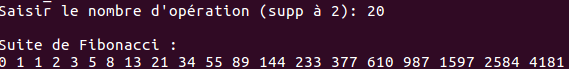
afficher(tab(i));

**FinPour**

**Fin**



**Exemple calcul de la suite de Fibonacci avec un vecteur :**



**Complexité :**

Affichage = n

Affectation = n+2

Addition = n-1

## Fibonacci avec des Variables

**Algorithme :**

**Données :** i, n, a, b, c  : entiers;

**Début**

// Lecture du nombre d’opération

lire(n) ;

// Initialisation

a ← 0 ;

afficher(a);

b ← 1;

afficher(b);

// Calcul de la suite de Fibonacci et affichage

**Pour** i allant de 2 à n **Faire**

c ← a+b;

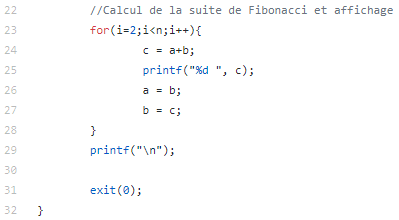
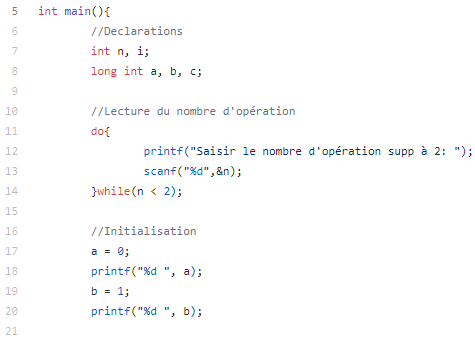
afficher(c);

a ← b;

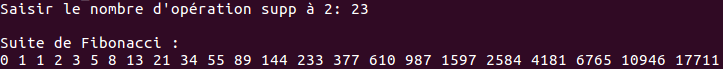
b ← c;

**FinPour**

**Fin**



**Exemple calcul de la suite de Fibonacci avec des variables :**



**Complexité :**

Affichage = n + 1

Affectation = 3n-1

Addition = n-1

## Majorité d’un tableau

**Algorithme :** majoritéTableau

**Données :** i, n, nb, compteur, tmp1, tmp2 : entiers;

\* tab : tableau d’entiers;

**Début**

// Lecture de la taille du tableau

lire(n);

// Initialisation

tab ← allouer(taille(entier)\*n);

// Lecture du tableau déjà trié

**Pour** i allant de 1 à n **Faire**

lire(tab(i));

**FinPour**

// Affichage du tableau

**Pour** i allant de 1 à n **Faire**

afficher(tab(i));

**FinPour**

// Calcul du majoré

tmp2 ← tab(0);

compteur ← 0;

tmp1 ← 1;

**Pour** i allant de 2 à n **Faire**

**Si** tmp2 != tab(i) **Alors**

**Si** tmp1 > compteur **Alors**

compteur ← tmp1;

nb ← tmp2;

**FinSi**

tmp1← 1;

tmp2 ← tab(i) ;

**Sinon**

tmp1 ← tmp1+1;

**FinSi**

**FinPour**

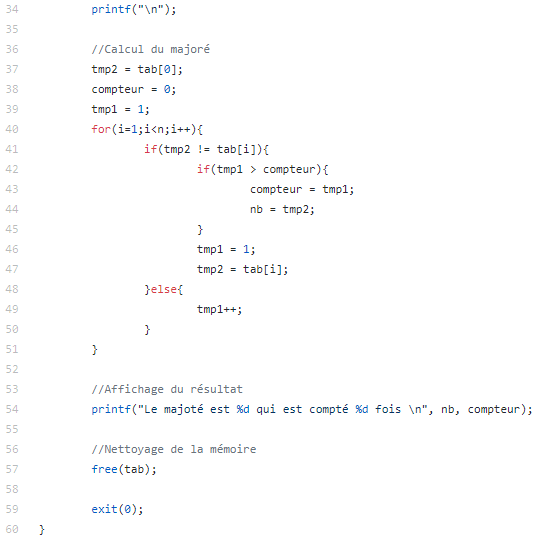
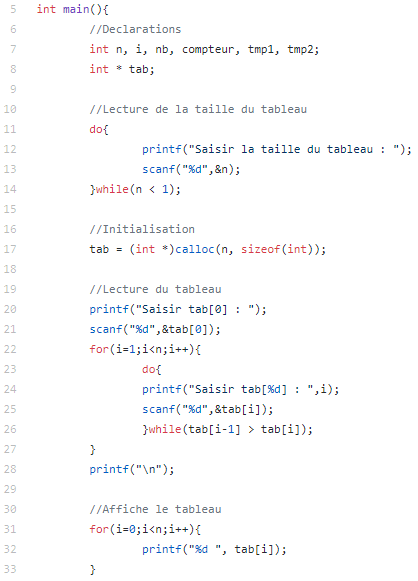
// Affichage du résultat

afficher(nb, compteur);

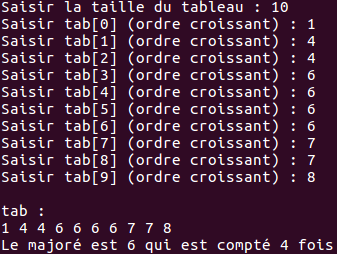
// Nettoyage de la mémoire

liberer(tab);

**Fin**



**Exemple calcul du majoré :**



**Complexité :**

Affichage = n+1

Affectation = 4n ou 2n+2 ou n+3

Lecture = n+1

## Conversion d’un nombre base 10 en nombre binaire

**Algorithme :** conversionBinaire

**Données :** i, n, nb, tmp : entiers;

\* v : tableau d’entiers;

**Début**

// Lecture du nombre à convetir

lire(nb);

// Calcul de la taille du tableau

tmp ← nb;

i ← 0 ;

**TantQue** tmp > 0 **Faire**

tmp ← tmp – puissance(2, i);

i ← i+1;

**FinTantQue**

n ← i;

// Initialisation du tableau v

v ← allouer(taille(entier)\*n);

// Calcul du nombre binaire

**Pour** i descendant de n à 1 **Faire**

**Si** (nb % 2) = 0 **Alors**

v(i) ← 0;

**Sinon**

v(i) ← 1;

**FinSi**

nb ← nb div 2 ;

**FinPour**

// Affichage du tableau v

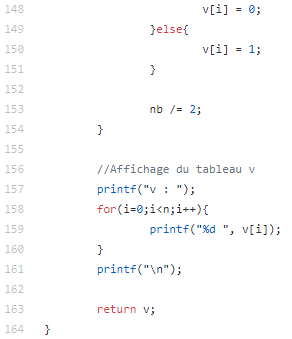
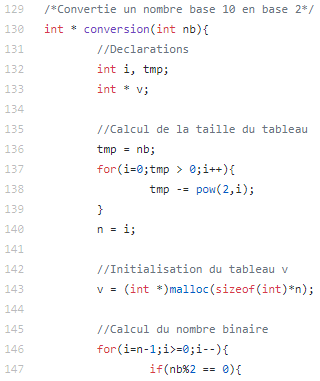
**Pour** i allant de 1 à n **Faire**

afficher(v(i));

**FinPour**

retourner(v);

**Fin**



**Exemple de conversion d’un nombre base 10 en binaire :**



**Complexité :**

Affichage = n

Affectation = 2n+2k+4 (k nb de tour de la boucle tant que)

Lecture = 1

# Algorithmes de Tri

## Tri via un Arbre

On utilise un arbre et le parcours Infiné pour trier un tableau. Chaque valeur est ajouté dans l’arbre selon ses rapports avec les autres nœuds, de façon récursive.

L’arbre est ensuite lu, permettant d’afficher les valeurs triées du tableau.

**Algorithme :** triViaArbre

**Données :** \*a : Structure Arbre;

i : Entier;

\*b : Structure Noeud;

\*T : Tableau d’entiers ;

**Début**

creerArbre(a) ;

**Pour** I **allant de** 1 à taille(T) **Faire**

**Si** a->racine == NULL **Alors**

allouer(b)

a->racine ← b;

**Sinon**

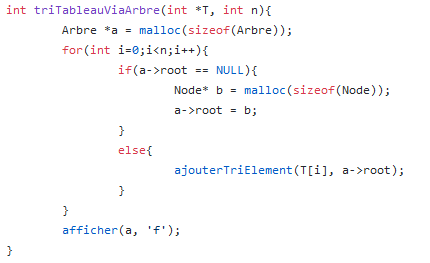
ajouterTriElement(T[i], a->root);

**FinSi**

**FinPour**

afficher(a, 'f');

**Fin**

****

### Tri de l’arbre

**Algorithme :** ajouterTriElement

**Données :** \*a : Structure Arbre;

i, ecart1, ecart2, tmp : Entier;

\*b,\*c : Structure Noeud;

\*T : Tableau d’entiers ;

**Début**

allouer(b);

b->valeur ← n;

**Si** c->fg == NULL **Alors**

c->fg ← b;

b->top ← c;

**Sinon** Si c->rgt == NULL **Alors**

c->fd ← b;

b->top ← c;

//puis inversion si valeur plus petit qu'à gauche

**Si** c->fd->valeur < c->fg->valeur **Alors**

tmp ← c->fd->valeur;

c->fd->valeur ← c->fg->valeur;

c->fg->valeur ← tmp;

**FinSi**

**Sinon**

**Si** c->fg->valeur < n **Alors**

ajouterTriElement(n, c->fg); //ajout recursif à gauche

**Sinon** Si c->fd->valeur < n **Alors**

ajouterTriElement(n, c->fd); //ajout recursif à droite

**Sinon**

ecart1 = n-c->fg->valeur;

ecart2 = n-c->fd->valeur;

**Si** ecart1>ecart2 **Alors** //ajout récursif à droite avec inversion de la valeur à insérer

tmp ← c->fd->valeur;

c->fd->valeur ← n;

ajouterTriElement(tmp, c->fd);

**Sinon**

//ajout récursif à gauche avec inversion de la valeur à insérer

tmp ← c->fg->valeur;

c->fg->valeur ← n;

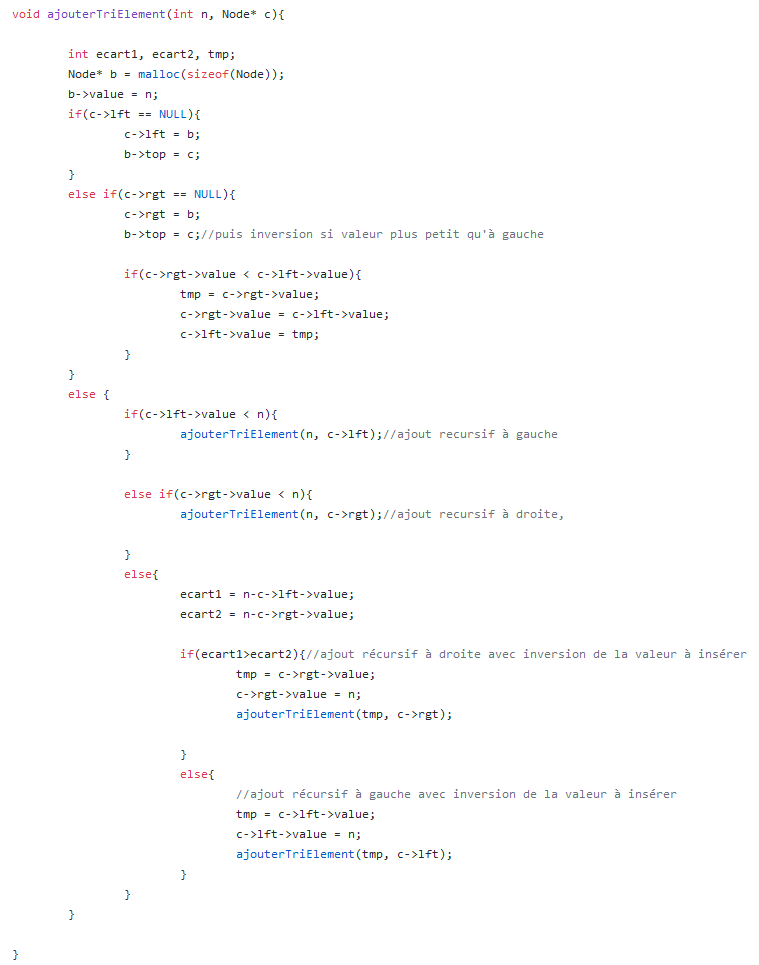
ajouterTriElement(tmp, c->fg);

**FinSi**

**FinSI**

**FinSi**

**Fin**

****

## Tri à bulle

**Algorithme :** trierBulle

**Données :**

\*T : Tableau d’entiers

stop : Booléen

i, valtmp : Entier

**Début**

stop ← Faux;

**TantQue** stop == Faux **Faire**

i = 0;

stop ← Vrai;

**TantQue** i < taille(T)-1 **Faire**

**Si** T[i] > T[i+1] **Alors**

valtmp ← T[i+1];

T[i+1] ← T[i];

T[i] ← valtmp;

stop ← Faux;

**FinSi**

i ← i+1;

**FinTantQue**

**FinTantQue**

**Fin**

****

****

**Complexité** : n² dans le pire, n au mieux

## Tri par Sélection

**Algorithme :** trierBulle

**Données :**

\*T : Tableau d’entiers

stop : Booléen

i,j, tmp,min : Entier

**Début**

**Pour** i allant de 0 à taille(T) – 1 **Faire**

min ← i;

**Pour** j allant de i + 1 à taille(T) **Faire**

**Si** T[j] < T[min] **Alors**

min ← j ;

**FinSi**

**FinPour**

**Si** min != i **Alors**

tmp ← T[min] ;

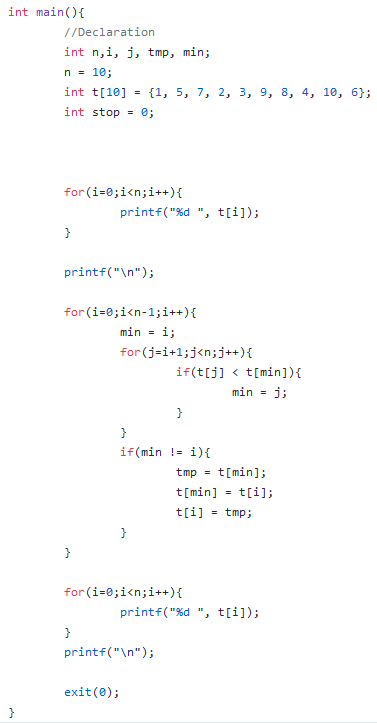
T[min] ← T[i] ;

T[i] = tmp ;

**FinSi**

**FinPour**

**Fin**

****



**Complexité** : n²

# Algorithmes de probabilité

## Simuler un jeu de tennis

**Algorithme :** simuTennis

**Données :** score1, score2 : entiers;

p : réél;

avantage : booléen;

**Début**

// Lecture de la probabilité de gagner

lire(p) ;

// Initialisation

score1 ← 0 ;

score2 ← 0 ;

avantage ← faux ;

// Simulation du jeu

**TantQue** (score1 != 4 **ET** score2 != 4 **ET**!avantage) **OU**  (score1 != 5 **ET** score2 != 5 **ET** avantage) **Faire**

**Si** score1 < 3 | score2 < 3 **Alors**

**Si** aléatoire() < p **Alors**

score1 ← score1+1;

**Sinon**

score2 ← score2+1;

**FinSi**

**Sinon**

avantage ← vrai ;

**Si** aléatoire() < p **Alors**

**Si** score2 = 4 **Alors**

score2 ← 3;

**Sinon**

score1 ← score1+1 ;

**FinSi**

**Sinon**

**Si** score1 = 4 **Alors**

score1 ← 3;

**Sinon**

score2 ← score2+1 ;

**FinSi**

**FinSi**

**FinSi**

**FinTantQue**

// Affiche si on a gagné ou perdu

**Si** score1 = 4 | score1 = 5 **Alors**

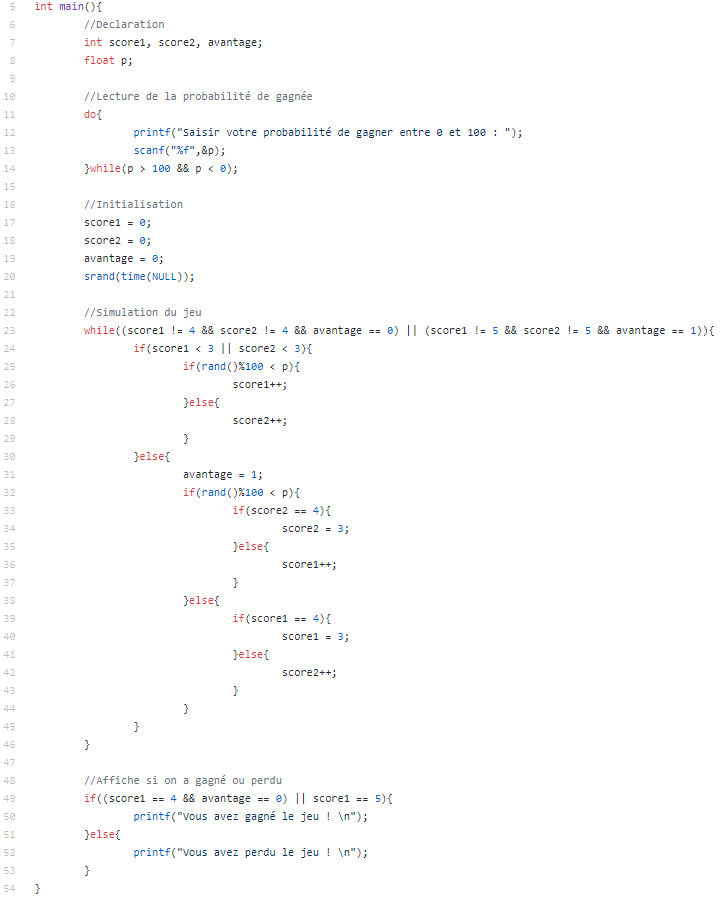
afficher(“vous avez gagné le jeu !“) ;

**Sinon**

afficher(“vous avez perdu le jeu !“) ;

**FinSi**

**Fin**



**Exemple de jeu de tennis :**









**Complexité :**

Affichage = 1

Lecture = 1

# Algorithmes de formalisme de pointeur

## Copier une chaîne de caractère

**Algorithme :** copieChaine

**Données :** n : entier;

\* p1, \* p2 : caractères;

tab(100) : tableau statique de caractères;

\* tab\_copy : tableau de caractères;

**Début**

// Lecture de la chaine de caractère (max 100)

lire(tab);

// Calcul de la taille de la chaine

p1 ← tab;

**TantQue** \*p1 != ‘\0’ **Faire**

p1 ← p1+1;

**FinTantQue**

n ← p1-tab;

// Initialisation

tab\_copy ← allouer(taille(caractère));

// Copie du tableau

p1 ← tab;

p2← tab\_copy;

**TantQue** \*p1 != ‘\0’ **Faire**

\*p2 ← \*p1;

p1 ← p1+1;

p2 ← p2+1;

**FinTantQue**

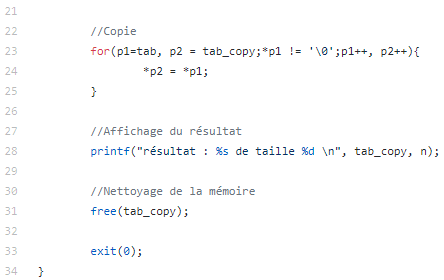
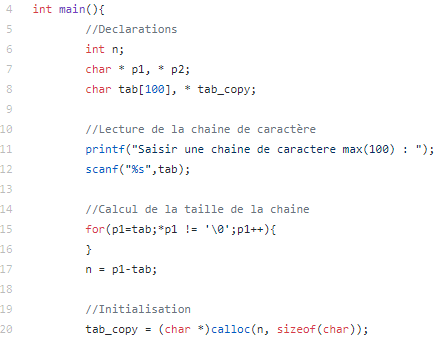
// Affichage du résultat

afficher(tab\_copy);

// Nettoyage de la mémoire

liberer(tab\_copy);

**Fin**



**Exemple de copie de chaine de caractère :**



**Complexité :**

Affichage = n

Affectation = 4n+5

Lecture = 1

## Supprimer l'occurrence

**Algorithme :** supprimerOccurence

**Données :** n, nb : entiers;

\* p1, \* p2 : entiers;

\* tab : tableau d’entiers;

**Début**

// Lecture de la taille du tableau

lire(n);

// Lecture de l’occurence

lire(nb);

// Initialisation

tab ← allouer(taille(entier));

// Lecture du tableau

**Pour** p1 allant de tab à (tab+n) **Faire**

lire(p1);

**FinPour**

// Affichage du tableau

**Pour** p1 allant de tab à (tab+n) **Faire**

afficher(\*p1);

**FinPour**

// Supprimer toutes les occurences

p2← tab;

**Pour** p1 allant de tab à (tab+n) **Faire**

\*p2← \*p1;

**Si** \*p2 != nb **Alors**

p2 ← p2+1;

**FinSi**

**FinPour**

n← p2-tab;

// Affichage du résultat

**Pour** p1 allant de tab à (tab+n) **Faire**

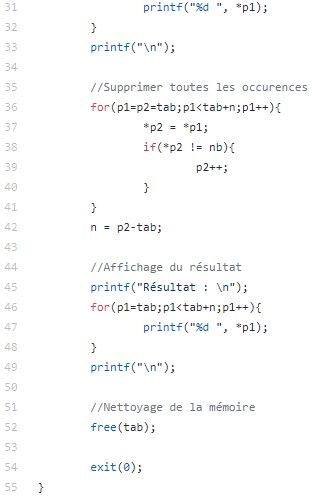
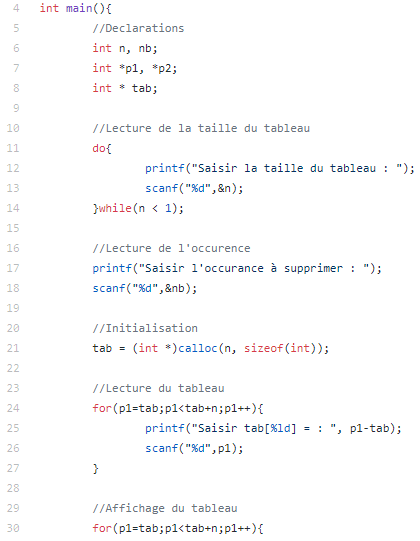
afficher(\*p1);

**FinPour**

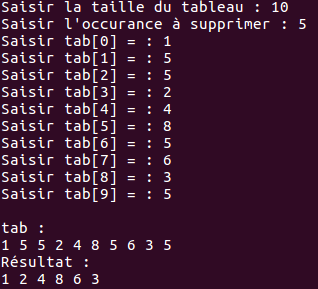
// Nettoyage de la mémoire

liberer(tab);

**Fin**



**Exemple supprimer l’occurrence :**



**Complexité :**

Affichage = 2n

Affectation = n+k+3 (k le nb d’occurrences supprimées)

Lecture = n+2

## Inverser les éléments d’un tableau

**Algorithme :** inversionTableau

**Données :** n, tmp : entiers;

\* p1, \* p2 : entiers;

\* tab : tableau d’entiers;

**Début**

// Lecture de la taille du tableau

lire(n);

// Initialisation

tab ← allouer(taille(entier));

// Lecture du tableau

**Pour** p1 allant de tab à (tab+n) **Faire**

lire(p1);

**FinPour**

// Affichage du tableau

**Pour** p1 allant de tab à (tab+n) **Faire**

afficher(\*p1);

**FinPour**

// Inverser les éléments du tableau

p2← tab+n-1;

**Pour** p1 allant de tab à (tab+n/2) **Faire**

tmp ← \*p1;

\*p2← \*p1;

\*p2 ← tmp;

p2 ← p2-1;

**FinPour**

// Affichage du résultat

**Pour** p1 allant de tab à (tab+n) **Faire**

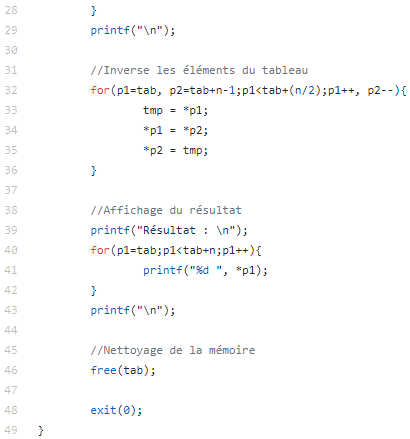
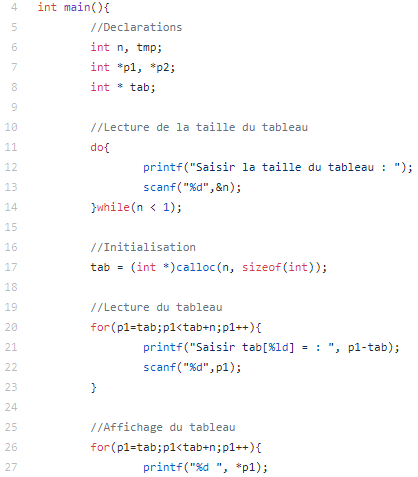
afficher(\*p1);

**FinPour**

// Nettoyage de la mémoire

liberer(tab);

**Fin**



**Exemple inversion des éléments d’un tableau :**



**Complexité :**

Affichage = 2n

Affectation = 2n+2

Lecture = n+1

## Palindrome

Un Palindrome est un mot consistant des mêmes caractères même s’il est inversé. Par exemple, Bob à l’envers donne Bob, tout comme Elle donne Elle.

**Algorithme :** chaineEstPalindrome

**Données :** n : entier

\* p1, \* p2 : char

tab(30) : tableau de caractère

test : booléen

**Début**

// Initialisation

test ← 1;

// Lecture de la chaine de caractère

lire(tab);

// Affichage du tableau

p1 ← tab;

**TantQue** \*p1 != ‘\0’ **Faire**

afficher(\*p1);

p1 ← p1+1;

**FinTantQue**

// Calculer la taille de la chaine de caractère

n ← p1-tab;

// Test si c’est un Palindrome

p2← tab+n-1;

**Pour** p1 allant de tab à (tab+n/2) **Faire**

**Si** \*p1 != \*p2 **Faire**

test ← 0;

**FinSi**

p2 ← p2-1;

**FinPour**

// Affichage du résultat

**Pour** p1 allant de tab à (tab+n) **Faire**

afficher(\*p1);

**FinPour**

**Si** test = 0 **Faire**

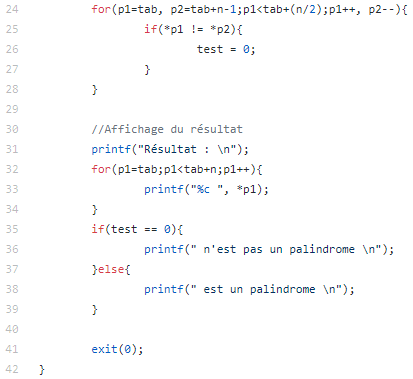
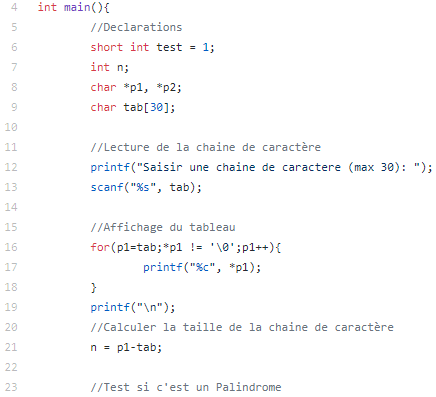
afficher(“n’est pas un palindrome“);

**Sinon**

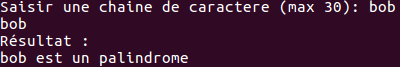
afficher(“est un palindrome“);

**FinSi**

**Fin**



**Exemple de test Palindrome :**



**Complexité :**

Affichage = 2n

Affectation = 1.5n+k+4 (k = le nombre de char != dans le « test »)

Lecture = 1

## Copier un tableau

**Algorithme :** copieTableau

**Données :** n : entier

\* p1, \* p2 : entiers

\* tab1, \* tab2 : tableaux d’entiers

**Début**

// Lecture de la taille du tableau

lire(n);

// Initialisation

tab1 ← allouer(taille(entier));

tab2 ← allouer(taille(entier));

// Lecture du tableau

**Pour** p1 allant de tab1 à (tab1+n) **Faire**

lire(p1);

**FinPour**

// Affichage du tableau

**Pour** p1 allant de tab1 à (tab1+n) **Faire**

afficher(\*p1);

**FinPour**

// Copier tab1 dans tab2

p2← tab2;

**Pour** p1 allant de tab1 à (tab1+n) **Faire**

**\***p2 = \*p1 ;

p2 ← p2+1 ;

**FinPour**

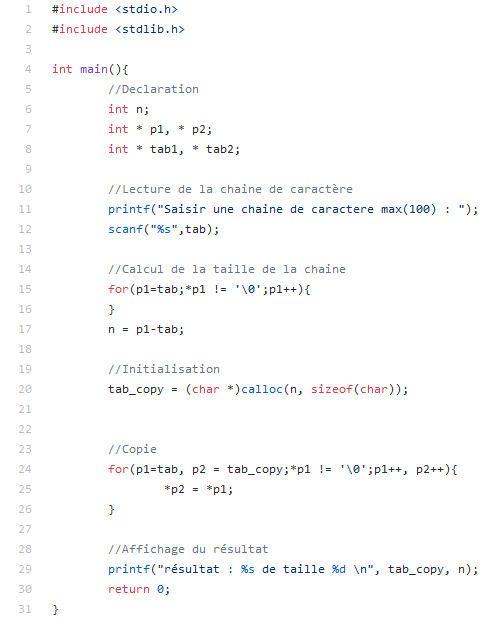
// Affichage du résultat

**Pour** p1 allant de tab2 à (tab2+n) **Faire**

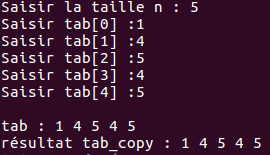
afficher(\*p1);

**FinPour**

**Fin**

****

**Exemple de copie de tableau :**



**Complexité :**

Affichage = 2n

Affectation = n+3

Lecture = n+1

# Algorithmes récursifs

## Ackerman récursif

**Algorithme :** AckermanRecursif

**Données :** n, m : entiers;

**Début**

// Fonction calcul d’Ackerman

**Si** m = 0 **Alors**

retourner(n+1);

**Sinon** Si n = 0 **Alors**

retourner(AckermanRecursif(m-1, 1));

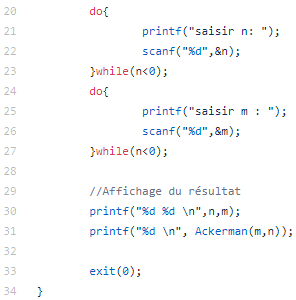
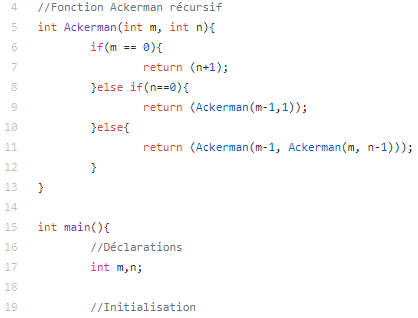
**Sinon**

retourner(AckermanRecursif(m-1, AckermanRecursif(m, n-1)));

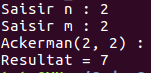
**FinSi**

**FinSi**

**Fin**



**Exemple Ackerman Récursif :**



## Tri d’un tableau par pivot (tri rapide) récursif

**Algorithme :** triRapide

**Données :** inf, sup, G, D, i, j, P, K : entiers;

\* T : tableau d’entiers;

**Début**

// Initialisation

G ← inf;

D ← sup;

i ← G;

j ← sup-1;

P ← T(sup);

// Calcul tri par pivot

**TantQue** i != j **Faire**

**TantQue** T(i) < P **ET** i != j **Faire**

i ← i+1;

**FinTantQue**

**TantQue** T(j) > P **ET** i != j **Faire**

j ← j+1;

**FinTantQue**

K ← T(i);

T(i) ← T(j);

T(j) ← K;

**FinTantQue**

**Si** T(i) > P **Alors**

K ← T(i);

T(i) ← T(sup);

T(sup) ← K;

**FinSi**

**Si** G < i **Alors**

triRapide(T, G, i);

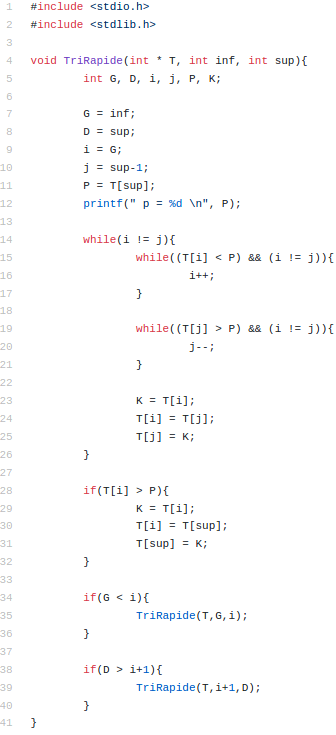
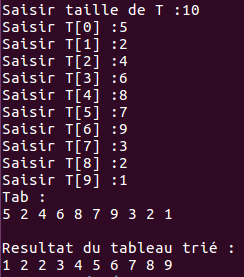
**FinSi**

**Si** D > i+1 **Alors**

triRapide(T, i+1, D);

**FinSi**

**Fin**

# Algorithmes de Pile

## Définition des structures

**Définir :** Structure cellule

**Début**

valeur : entier;

\* suivant : structure cellule;

**Fin**

**Nommer** Cellule

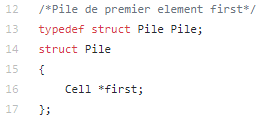
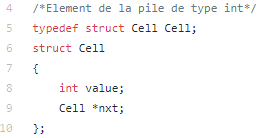
**Définir :** Structure pile

**Début**

\* premier : structure cellule;

**Fin**

**Nommer** Pile



## Créer une Pile

**Algorithme :** creerPile

**Données :** \* p : structure pile;

**Début**

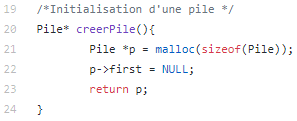
// Allocation et initialisation de la pile

p ← allouer(taille(structure pile));

p → premier ← NULL;

retourner(p);

**Fin**



## Empiler une Pile

**Algorithme :** empilerPile

**Données :** \* p : structure pile;

val : entier;

\* c : structure cellule;

**Début**

// Allocation et initialisation de la cellule

c ← allouer(taille(structure cellule));

c → valeur ← val;

c → suivant ← NULL;

// Empiler la pile

**Si** p != NULL **OU** c != NULL **Faire**

**Si** p → premier != NULL **Faire**

c → suivant ← p → premier;

p → premier ← c;

**Sinon**

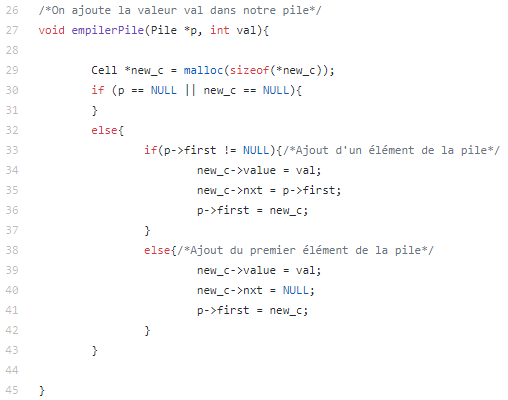
c → suivant ← NULL;

p → premier ← c;

**FinSi**

**FinSi**

**Fin**



## Dépiler une Pile

**Algorithme :** depilerPile

**Données :** \* p : structure pile;

val : entier;

\* c : structure cellule;

**Début**

// Initialisation

\* c ← p → premier;

// Depiler la pile

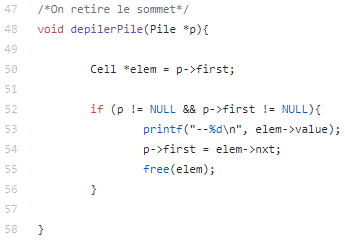
**Si** p != NULL **ET** p → premier != NULL **Faire**

p → premier ← c → suivant;

liberer(c);

**FinSi**

**Fin**



## Sommet d’une pile

**Algorithme :** sommet

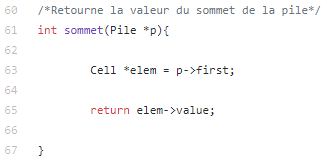
**Données :** \* p : structure pile;

**Début**

// Retourne le sommet de la pile

retourner(p → premier → valeur);

**Fin**



## Déterminer si une pile est vide

**Algorithme :** pileEstVide

**Données :** \* p : structure pile;

test : booléen;

**Début**

// Initialisation

test ← faux;

// Test si la pile est vide

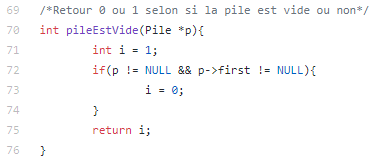
**Si** p != NULL **ET** p → premier != NULL **Faire**

p → premier ← c → suivant;

liberer(c);

**FinSi**

**Fin**



## Vider une Pile

**Algorithme :** viderPile

**Données :** \* p : structure pile;

**Début**

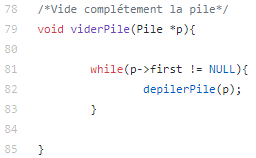
// Vide la pile

**TantQue** p → premier != NULL **Faire**

depiler(p);

**FinTantQue**

**Fin**



## Afficher une Pile

**Algorithme :** afficherPile

**Données :** \* p, \* tmp : structure pile;

**Début**

// Affichage de la pile

**TantQue** p → premier != NULL **Faire**

empiler(tmp, sommet(p));

afficher(sommet(p));

depiler(p);

**FinTantQue**

**TantQue** tmp → premier != NULL **Faire**

empiler(p, sommet(tmp));

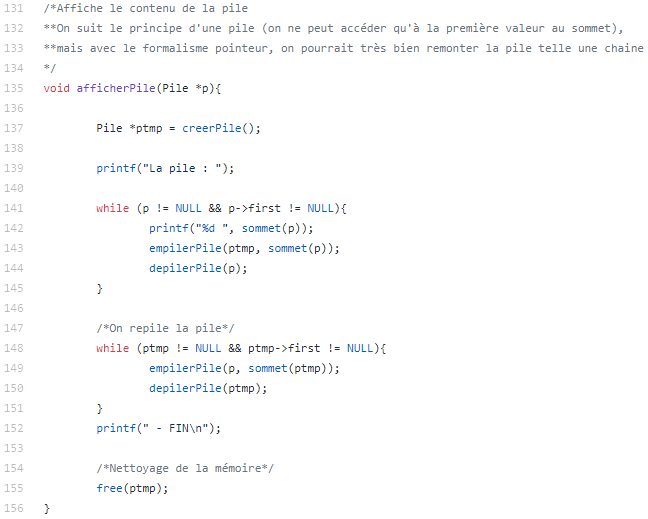
depiler(tmp);

**FinTantQue**

// Nettoyage de la mémoire

liberer(tmp);

**Fin**



## Egalité de deux Piles

**Algorithme :** estEgalePile

**Données :** \* p1, \* p2, \* tmp1, \* tmp2 : structure pile;

test : booléen;

**Début**

// Initialisation

tmp1 ← creerPile();

tmp2 ← creerPile();

test ← vraie;

// Test si les deux piles sont égales

**TantQue** p1 → premier != NULL **ET** p2 → premier != NULL **Faire**

empiler(tmp1, sommet(p1));

empiler(tmp2, sommet(p2));

**Si** sommet(p1) != sommet(p2)) **Faire**

test ← faux;

**FinSi**

depiler(p1);

depiler(p2);

**FinTantQue**

// On vérifie si la taille est différente

**Si** p1 → premier != NULL **OU** p2 → premier != NULL **Faire**

test ← faux;

**FinSi**

// On reempile les piles

**TantQue** tmp1 → premier != NULL **ET** tmp2 → premier != NULL **Faire**

empiler(p1, sommet(tmp1));

empiler(p2, sommet(tmp2));

depiler(tmp1);

depiler(tmp2);

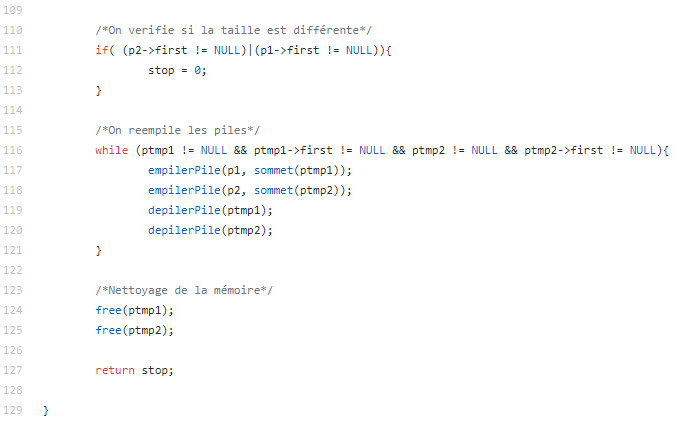
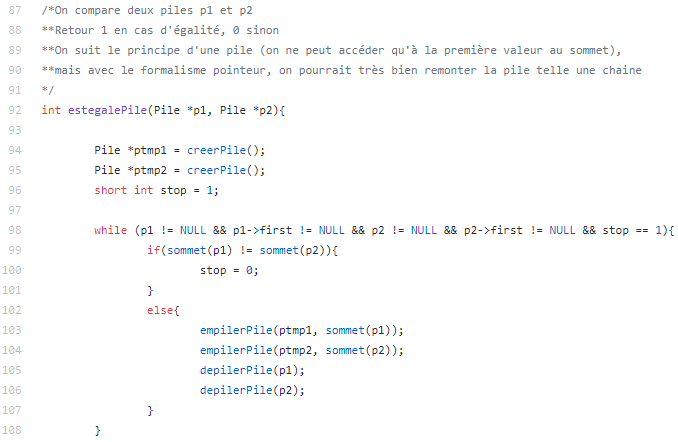
**FinTantQue**

// Nettoyage de la mémoire

liberer(tmp1);

liberer(tmp2);

**Fin**



## Trier une Pile

**Algorithme :** trierPile

**Données :** \* p, \* tmp1, \* tmp2 : structure pile;

min : entier;

**Début**

// Initialisation

tmp1 ← creerPile();

tmp2 ← creerPile();

// Tri la pile

**TantQue** p → premier != NULL **Faire**

min ← sommet(p);

**TantQue** p → premier != NULL **Faire**

**Si** sommet(p) < min **Faire**

min ← sommet(p);

**FinSi**

empiler(tmp1, sommet(p));

depiler(p) ;

**FinTantQue**

**TantQue** tmp1 → premier != NULL **Faire**

**Si** sommet(tmp1) != min **Faire**

empiler(p, sommet(tmp1));

**FinSi**

depiler(tmp1) ;

**FinTantQue**

empiler(tmp2, min);

**FinTantQue**

//On replace les valeurs dans p

**TantQue** tmp2 → premier != NULL **Faire**

empiler(p, sommet(tmp2));

depiler(tmp2);

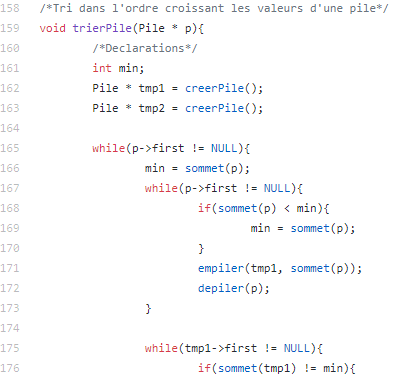
**FinTantQue**

// Nettoyage de la mémoire

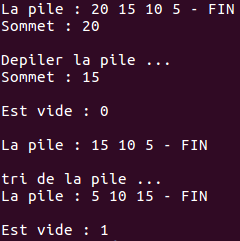
liberer(tmp1);

liberer(tmp2);

**Fin**



**Exemple de pile :**



## Ackerman avec Pile (Iteratif)

**Algorithme :** AckermanPile

**Données :** n, m : entiers;

\* p : Pile;

**Début**

// Initialisation

p ← creerPile();

empiler(p, m);

empiler(p, n);

// Fonction calcul d’Ackerman

**TantQue** **NON** pileEstVide(p) **Faire**

n← sommet(p);

depiler(p);

Si **NON** pileEstVide(p) **Alors**

m ← sommet(p);

depiler(p);

**Sinon**

Sortir;

**FinSi**

**Si** m = 0 **Alors**

empiler(p, n+1);

**Sinon** Si n = 0 **Alors**

empiler(p, m-1);

empiler(p, 1);

**Sinon**

empiler(p, m-1);

empiler(p, m) ;

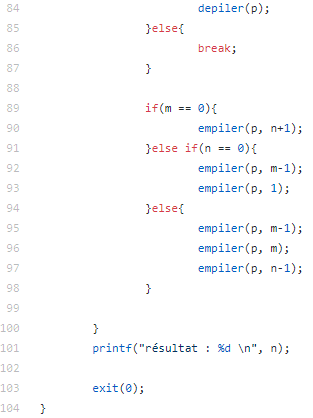
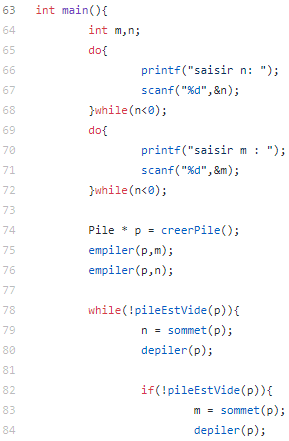
empiler(p, n-1) ;

**FinSi**

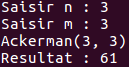
**FinSi**

**FinTantQue**

**Fin**



**Exemple de calcul d’Ackerman récursif :**



# Algorithmes de Liste

Afin de simplifier nos calculs et ajouter un meilleur contrôle sur le nombre d’éléments, nous avons ajouté un attribut « taille » en plus, s’incrémentant ou se décrémentant selon au fil des utilisations de la liste.

## Définition des structures

**Définir :** Structure élément

**Début**

valeur : entier;

\* prochain : structure élément;

**Fin**

**Nommer** élément\_liste

**Définir** Structure Liste

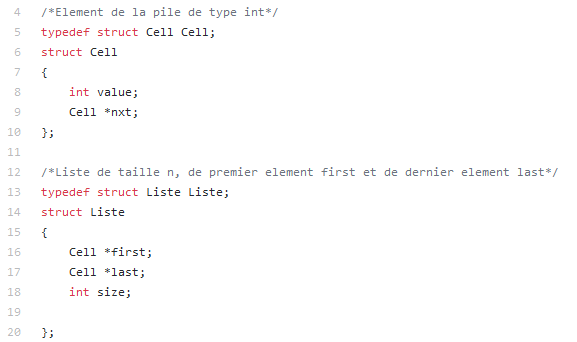
**Début**

\*premier, dernier : structure élément;

taille : entier;

**Fin**

**Nommer :** Liste



## Créer une liste

**Algorithme :** CreerListe

**Données :** \*l : structure liste;

**Début**

// Allocation de la liste

Allouer(l) ;

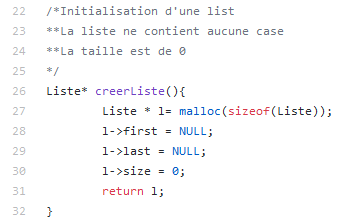
// Initialisation des pointeurs et de la taille

l->premier ← NULL ;

l-> dernier ← NULL ;

taille ← 0;

**Fin**



## La liste est-elle vide ?

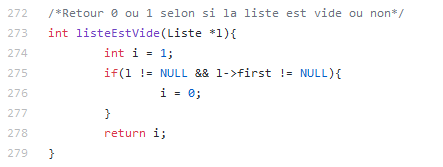
**Algorithme** : estVideListe

**Données** : \*l : liste;

**Début**

retourner(l->premier == NULL);

**Fin**

****

## Afficher la liste

**Algorithme :** afficherListe

**Données :** \*l : structure liste

\*e : structure élément

**Début**

e ← l->premier ;

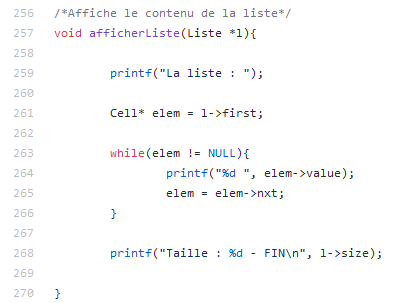
**TantQue** e != NULL **Faire**

Ecrire(e->valeur) ;

e ← e->prochain ;

**FinTantQue**

**Fin**



## Chercher un élément dans la liste

Grace à la taille, il est bien plus facile de se déplacer dans la liste et de gérer les cas où l’utilisateur saisit des valeurs invalides. Ceci nous évite notamment de faire un parcours de toute la boucle jusqu’à trouver la valeur.

Retourne la n-ième valeur d'une liste. Retourne la dernière valeur si l’utilisateur rentre un n supérieur à la taille de la liste. S’il saisit une valeur inférieure à 0, parcours la liste dans le sens inverse !

**Algorithme :** chercherElementListe

**Données :** \* l : structure liste

n, i, j : entiers

\* e : structure élément

**Début**

i ← 0

j ← n

e ← l-> premier ;

**Si** e != NULL **Alors**

//Si n est plus grand que la taille, on retourne la dernière valeur de la liste

**Si** j > l->taille **Alors**

**retourner** l->dernier;

**Sinon**

**Si** j<0 **Alors**

j ← (l->taille-n)%l->taille;//On augmente j pour coller à la taille et parcourir dans le sens inverse

**FinSi**

**Tant que** i < j **Faire**

e ← e->prochain;

i ← I + 1;

**FinTantQue**

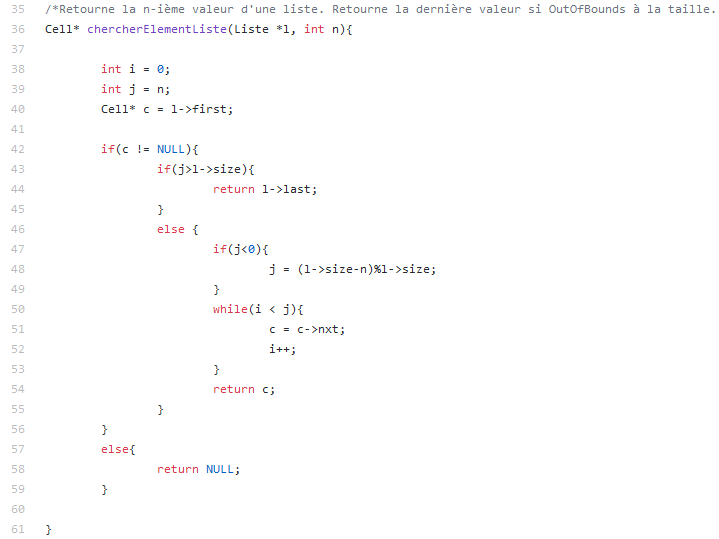
**retourner** e;

**FinSi**

**Sinon**

retourner NULL; //l est vide

**FinSi**

****

## Ajouter en dernière position

**Algorithme :** ajouterDernierElementListe

**Données :**

\*l : Liste

valeur : Entier

\*nouvel\_element,\* tmp : Structure Élément

**Début**

\*nouvel\_element ← malloc(sizeof(\*nouvel\_element));

**Si** l != NULL ET nouvel\_element != NULL **Alors**

**Si** l->premier != NULL **Alors** //Ajout d'un élément lambda

//On définit la nouvelle case

nouvel\_element->valeur ← valeur;

nouvel\_element->prochain ← NULL;

//On redéfinit la dernière case

tmp ← l->dernier;

tmp->prochain ← nouvel\_element ;

//On définit la nouvelle case comme dernière case

l->dernier ← nouvel\_element;

//On augmente la taille

l->taille ← l->taille + 1;

**Sinon**

//Ajout du premier élément de la pile

nouvel\_element->valeur ← valeur ;

nouvel\_element->prochain ← NULL;

l->premier ← nouvel\_element ;

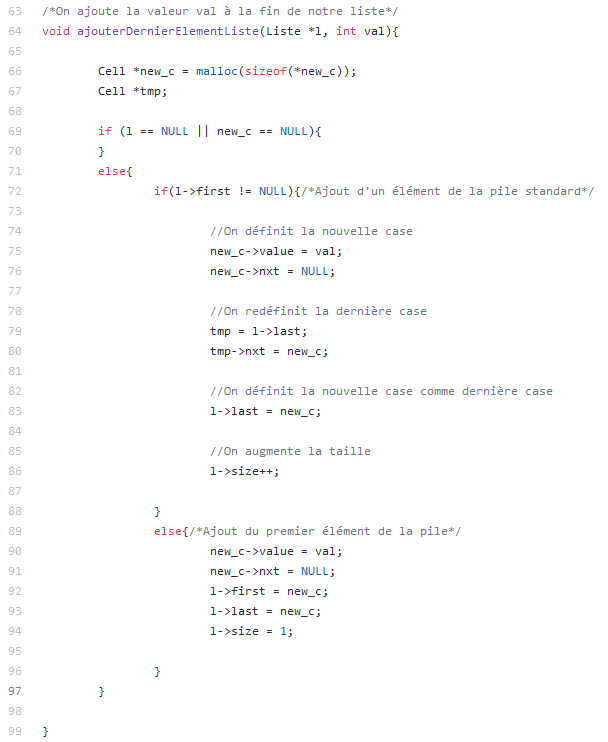
l->dernier ← nouvel\_element;

l->taille ← 1;

**FinSi**

**FinSi**

**Fin**



## Ajouter la valeur à la nième position

**Algorithme :** ajouterElementListe

**Données :**

\*l : Liste

valeur, n,j: entier

\*nouvel\_element,\* tmp : Structure Élément

**Début**

**Si** l != NULL ET nouvel\_element != NULL **Alors**

**Si** l->premier != NULL **Alors** //Ajout d'un élément de la liste standard

**Si** n<l->taille et n>0 **Alors**

//On définit la nouvelle case

nouvel\_element->valeur ← valeur;

nouvel\_element->prochain = chercherElementListe(l, n+1);

//On redéfinit la case n-1

tmp ← chercherElementListe(l, n-1);

tmp->prochain ← nouvel\_element;

//On augmente la taille

l->taille ← l->taille +1 ;

**Sinon**

**Si** n>= l->taille **Alors**

ajouterDernierElementListe(l, valeur);

**Sinon**

j ← (l->taille-n)%l->taille;

ajouterElementListe(l,valeur, j);

**FinSi**

**FinSi**

**Sinon** //Ajout du premier élément de la pile

nouvel\_element->valeur = valeur;

nouvel\_element->prochain = NULL;

l->premier ← new\_c;

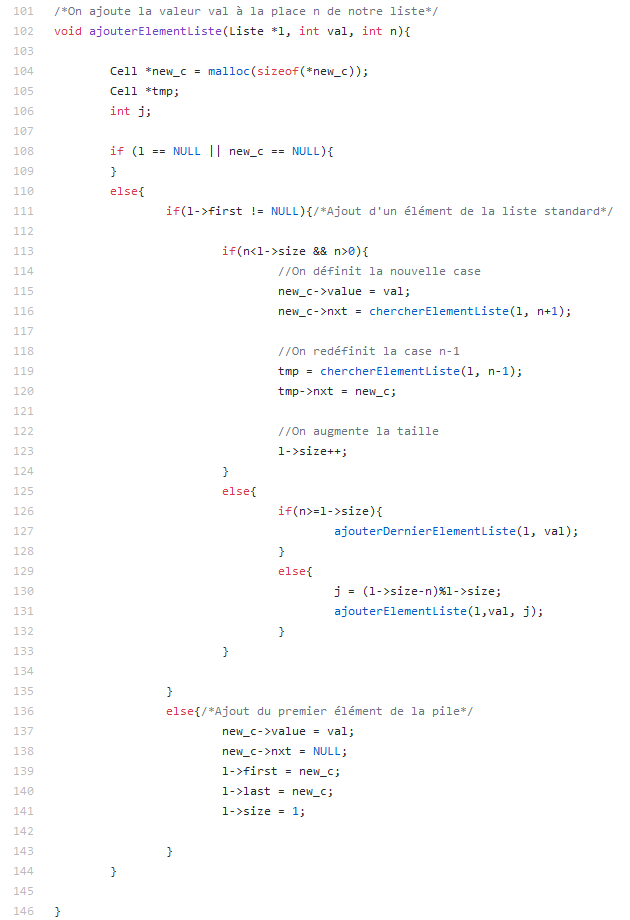
l->dernier ← new\_c;

l->taille ← l->taille + 1;

**FinSi**

**FinSi**

**Fin**

****

## Retirer la valeur à la dernière position

**Algorithme :** supprimerDernierElementListe

**Données :**

\*l : Liste

\*elem : Structure Élément

**Début**

**Si** l->premier != NULL **Alors**

**Si** l->taille > 1 **Alors**

elem ← chercherElementListe(l, l->taille-2); //Retourne l'avant-dernière case

elem->prochain ← NULL; //Suppression du dernier élément

libérer(l->dernier) ;

l->taille ← l->taille - 1; //Réduction de la taille

l->dernier ← elem; //Redéfinition de la dernière case

**Sinon**

//La liste ne contient qu'une seule case

elem ← l->premier; //Retourne la seule et unique case

l->premier ← NULL;

l->dernier ←NULL;

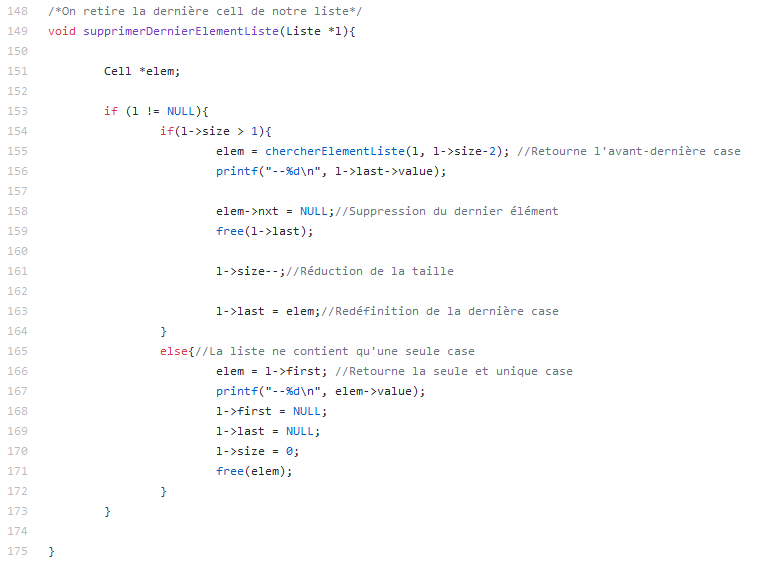
l->taille ← 0;

libérer(elem);

**FinSi**

**FinSi**

**Fin**



## Retirer la valeur à la position n

**Algorithme :** supprimerElementListe

**Données :**

\*l : Liste

\*elem1, \*elem2 , \*elem3  : Structure Élément

i: Entier

**Début**

**Si** l->premier != NULL **Alors**

**Si** l->taille > 1 **Alors**

**Si** n<l->taille & n > 0 **Alors** //Suppression lambda

elem1 = chercherElementListe(l, n); //Retourne l'élément n

elem2 = chercherElementListe(l, n-1); //Retourne l'élément n-1

elem3 = elem1->prochain ; //Retourne l'élément n+1

elem2->prochain ← elem3; //On saute l'élement n à supprimer

l->taille ← l->taille; //Réduction de la taille

libérer(elem1);

**Sinon**

**Si** n>=l->taille **Alors** //n est plus grand que la taille, on supprime le dernier élément

supprimerDernierElementListe(l);

**Sinon**

**Si** n == 0 **Alors** //premier élément de la liste à supprimer

elem1 ← chercherElementListe(l, 0); //Retourne l'élément n

elem2 ← elem1->prochain ; //Retourne l'élément n+1

l->premier ← elem2;

l->taille ← l->taille; //Réduction de la taille

libérer(elem1);

**Sinon** //n négatif, on parcours dans le sens inverse

j ← (l->taille-n)%l->taille;

supprimerElementListe(l, j); //Appel récursif

**FinSi**

**FinSi**

**FinSi**

**Sinon** //La liste ne contient qu'une seule case : on la supprime donc

elem1 ← l->premier; //Retourne la seule et unique case

l->premier← NULL;

l->dernier ←NULL;

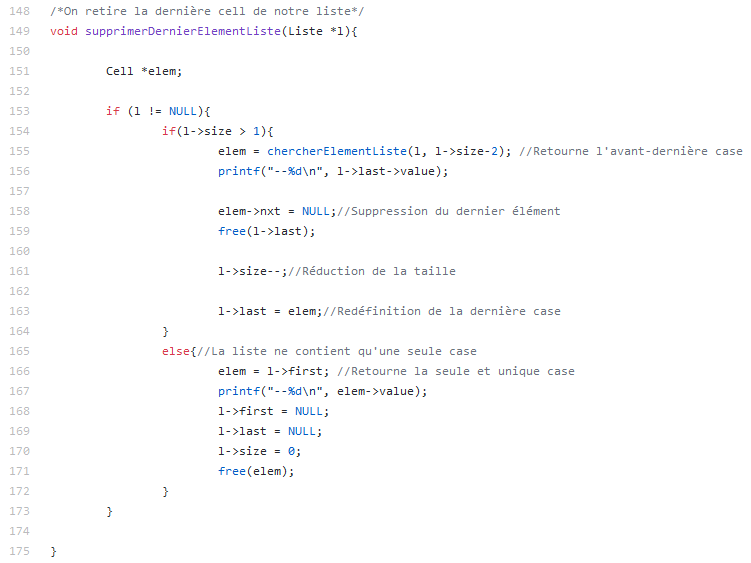
l->taille ← 0;

libérer(elem1);

**FinSi**

**FinSi**

**Fin**



## Retirer la valeur (première occurrence)

**Algorithme :** supprimerValeurListe

**Données :**

\*l : Liste

\*elem : Structure Élément

i, retour, valeur : Entiers

**Début**

elem ← l->premier;

i = 0;

retour = 0;

//On se déplace vers l’occurence

**TantQue** elem != NULL ET elem->valeur != valeur **Faire**

elem = elem->prochain;

I ← I +1;

**FinTantQue**

//Si fin de la liste

**Si** i<l->taille **Alors**

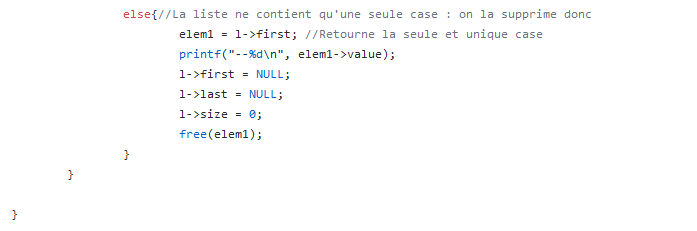
supprimerElementListe(l, i);

retour ← 1;

**FinSi**

retourner retour;//On a bien supprimé la valeur

**Fin**



## Afficher Liste

**Algorithme :** afficherListe

**Données :**

\*l : Liste

\*elem : Structure Élément

**Début**

elem ← l->premier;

Ecrire(“La liste :“) ;

**Tant que** elem != NULL **Faire**

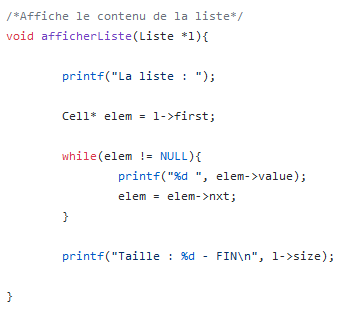
Ecrire(“ “  + elem->valeur + “ “);

elem ← elem->prochain;

**FinTantQue**

Ecrire(“Taille : “ + l->taille + “ - FIN“);

**Fin**

****

## La liste est vide ?

**Algorithme :** listeEstVide

**Données :**

\*l : Liste

i : Booléen

**Début**

i ← False;

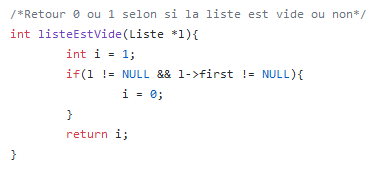
**Si** l != NULL et l->first != NULL **Alors**

i ← True;

}

retourner i;

**Fin**

****

## Vider la liste

**Algorithme :** viderListe

**Données :**

\*l : Liste

i : Booléen

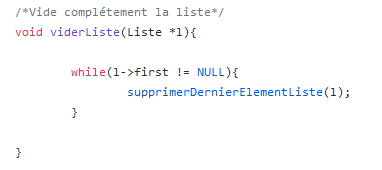
**Début**

**TantQue** l->premier != NULL **Alors**

supprimerDernierElementListe(l);

**FinTantQue**

**Fin**

****

## Egalité de deux listes

**Algorithme :** estEgaleListe

**Données :**

\*l1, \*l2 : Liste

stop : Booléen

\*elem1, \*elem2 : Structure Élément

**Début**

stop ← True ;

elem1 ← l1->premier;

elem2 ← l2->premier;

//On verifie si la taille est différente

**Si** l1->taille != l2 ->taille **Alors**

stop ← False

**FinSi**

**TantQue** elem1 != NULL ET elem2 != NULL ET stop == True **Faire**

**Si** elem1->value != elem2->value **Alors**

stop ← False

**Sinon**

elem1 ← elem1->prochain;

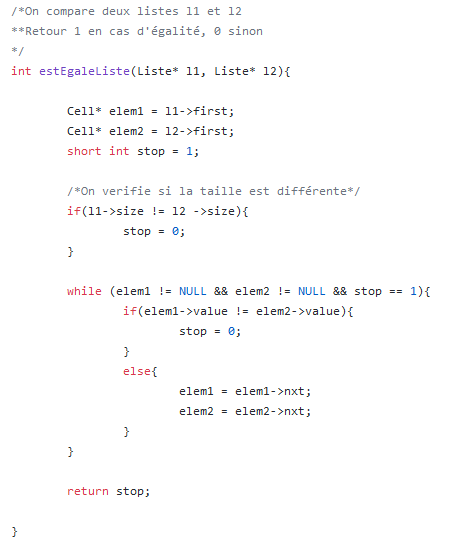
elem2 ← elem2->prochain;

**FinSi**

**FinTantQue**

retourner stop;

**Fin**

****

## Construire une liste binaire à partir d’un tableau

**Algorithme :** constructionListeBinaire

**Données :** n, i : entiers;

\* v : tableau d’entiers;

\* l : Liste;

**Début**

//Initialisation

l ← creerListe();

//On verifie si la taille est différente

**Pour** i allant de 1 à n **Faire**

ajouterDernierElementListe(l, v(i));

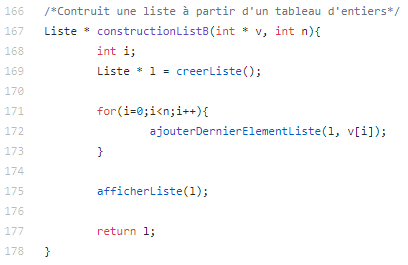
**FinPour**

//Affichage de la Liste

afficherListe(l);

retourner(l);

**Fin**



## Incrémentation de 1 une liste binaire

**Algorithme :** incrementationListeBinaire

**Données :** counter : entier;

\* v : tableau d’entiers;

\* l : Liste;

\* c : Cellule;

**Début**

//Incrementation

**Si** l != NULL **ET** l → premier != NULL **Alors**

counter ← l → taille-2;

c ← l → dernier;

**TantQue** c != NULL **ET** counter >= 0 **ET** c → valeur != 0 **Faire**

c → valeur ← 0;

c ← chercherElementListe(l, counter);

counter ← counter-1;

**FinTantQue**

**Si** c != NULL **ET** counter >= 0 **Alors**

c → valeur ← 1;

**Sinon**

ajouterDernierElementListe(l, 0);

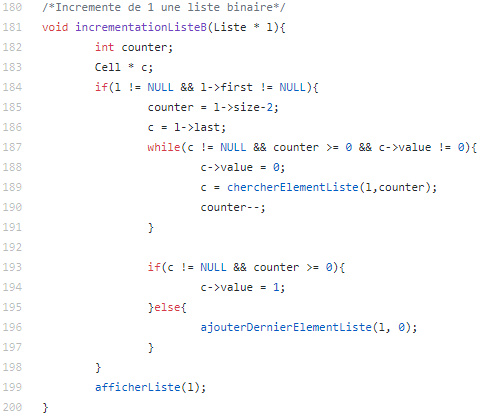
**FinSI**

**FinSI**

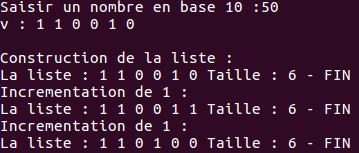
//Affichage de la Liste

afficherListe(l);

**Fin**



**Exemple liste binaire :**



## Trier une liste

**Algorithme :** trierListe

**Données :**

\*l : Liste

stop: Booléen

valtmp Entier

\*elem : Structure Elément Liste

**Début**

stop ← Faux;

**TantQue** stop == Faux **Faire**

elem ← l->premier;

stop ← Vrai;

**TantQue** elem->prochain != NULL) **Faire**

**Si** elem->prochain->valeur < elem->valeur **Alors**

valtmp ← elem->prochain->valeur;

elem->prochain->valeur ← elem->valeur;

elem->valeur ← valtmp;

stop ← Faux;

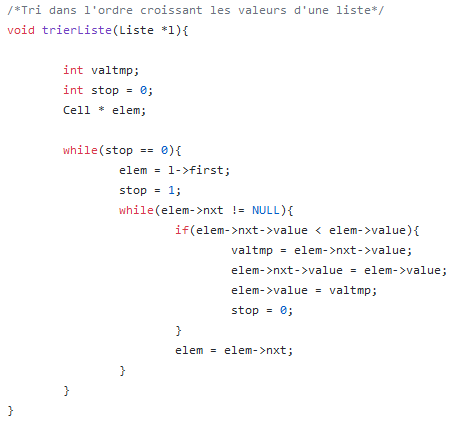
**FinSi**

elem ← elem->prochain;

**FinTantQue**

**FinTantQue**

**Fin**



## Fusionner deux listes triées

**Algorithme :** fusionnerListes

**Données :**

\*l1, \*l2, \*l3 : Liste

i : Booléen

\*elem1, \*elem2: Structure Element Liste

**Début**

l3 ← creerListe();

elem1 C l1->premier;

elem2 ← l2->premier;

**TantQue** elem1 != NULL || elem2 != NULL **Faire**

**Si** elem1 != NULL **Alors**

**Si** elem2!=NULL **Alors**

**Si** elem1->valeur < elem2->valeur **Alors**

ajouterDernierElementListe(l3, elem1->valeur);

elem1 ← elem1->prochain ;

**Sinon**

ajouterDernierElementListe(l3, elem2->valeur);

elem2 ← elem2->prochain;

**FinSi**

**Sinon**

ajouterDernierElementListe(l3, elem1->valeur);

elem1 ← elem1->prochain;

**FinSi**

**Sinon**

ajouterDernierElementListe(l3, elem2->valeur);

elem2 ← elem2->prochain;

**FinSi**

**FinTantQue**

return l3;

**Fin**



## Extraire une chaine de la liste

**Algorithme :** extraireChaine

**Données :**

\*l : Liste

a,b,c : Booléen

elem : Structure Element Liste

**Début**

c ← 0;

**TantQue** elem != NULL **Faire**

**Si** c >= a && c <= b **Alors**

Ecrire(elem->valeur);

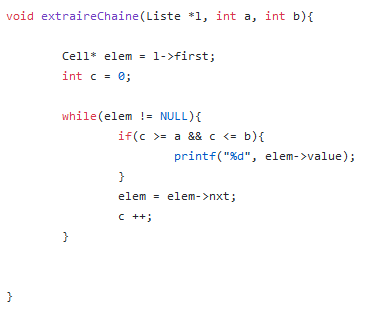
**FinSi**

elem ← elem->prochain;

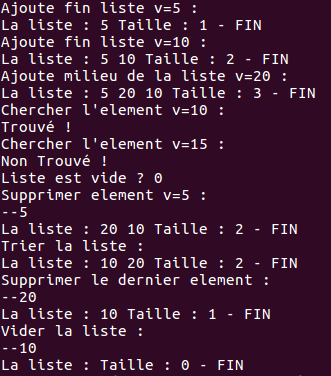
c ++;

**FinTantQue**

**Fin**



**Exemple d’une liste :**



# Algorithmes d’Arbre

**Note :** Les piles utilisées dans les algorithmes ci-dessous ne stockent pas des entiers mais des Nœuds

## Définition des structures

**Définir :** Structure noeud

**Début**

valeur : entier;

\* pere, fd, fg : structure noeud;

**Fin**

**Nommer** nœud\_arbre

**Définir** Structure Arbre

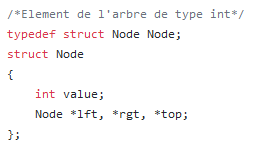
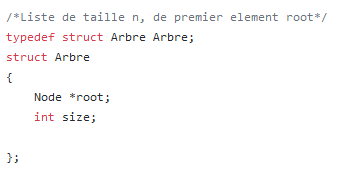
**Début**

\*racine : structure noeud;

taille : entier;

**Fin**

**Nommer :** Arbre



## Créer un Arbre

**Algorithme :** CreerArbre

**Données :** \*a : structure Arbre;

**Début**

// Allocation de l’arbre

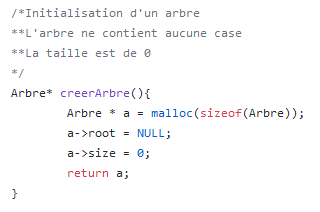
Allouer(a) ;

// Initialisation de la racine et de la taille

a->racine ← NULL ;

taille ← 0;

**Fin**

****

## L’arbre est-il vide ?

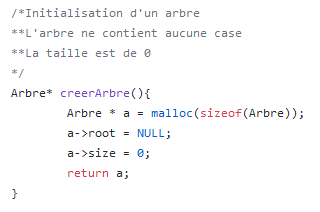
**Algorithme** : estVideArbre

**Données** : \*a : arbre;

**Début**

retourner(a->racine == NULL);

**Fin**



## Choix du parcours pour l’affichage

**Algorithme :** Afficher

**Données**

\*a : Structure Arbre

c : caractère ;

**Début**

**Selon** (c) :

**cas** ‘p’ :

afficherPrefixe(a->racine) ;

**fincas** ;

**cas** ‘i’ :

afficherInfine(a->racine) ;

**fincas**;

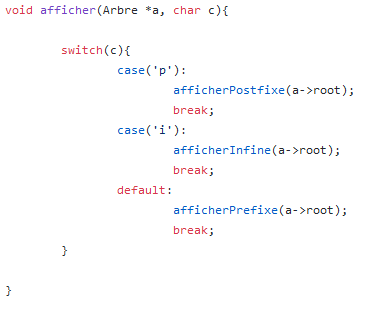
**Autrement** :

afficherSuffixe(a->racine) ;

**fincas** ;

**FinSelon**

**Fin**



### Parcours Prefixé

**Algorithme**: afficherPrefixe

**Données**:

\*n : structure nœud ;

**Début**

**Si** n != NULL **Alors**

Ecrire(e->valeur) ;

afficherPrefixe(e->fg) ;

afficherPrefixe(e->fd) ;

**FinSi**

**Fin**

### Parcours Infiné

**Algorithme**: afficherInfine

**Données**:

\*n : structure nœud ;

**Début**

**Si** n != NULL **Alors**

afficherInfine (e->fg) ;

Ecrire(e->valeur) ;

afficherInfine (e->fd) ;

**FinSi**

**Fin**

### Parcours Suffixe

**Algorithme**: afficherSuffixe

**Données**:

\*n : structure nœud ;

**Début**

**Si** n != NULL **Alors**

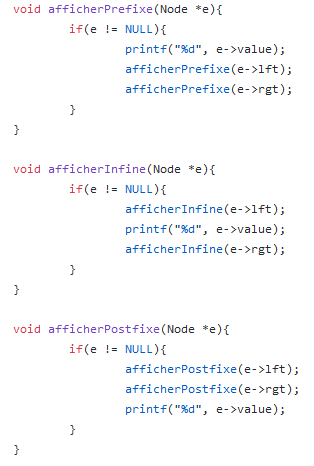
afficherSuffixe (e->fg) ;

afficherSuffixe (e->fd) ;

Ecrire(e->valeur) ;

**FinSi**

**Fin**

****

## Calcul de la hauteur

**Algorithme :** calculHauteur

**Données :**

\*a : structure Arbre

h : entier

**Début**

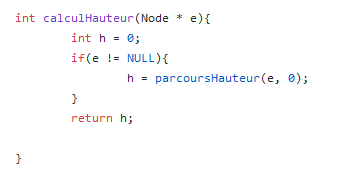
**Si**NON estVideArbre(a) **Alors**

h ← parcoursHauteur(a->racine, 0) ;

**FinSi**

**retourner** h ;

**Fin**



## Parcours pour le calcul de la hauteur

**Algorithme :** parcoursHauteur

**Données :**

\*n : structure Noeud

h, htmp1, htmp : entiers

**Début**

**Si** n != NULL **Alors**

htmp1 ← parcoursHauteur(n->fg, h+1) ;

htmp2 ← parcoursHauteur(n->fd, h+1) ;

**Si** htmp 2 > htmp1 **Alors**

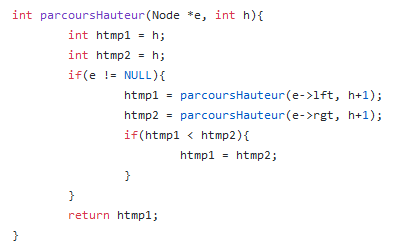
htmp1 ← htmp2 ;

**FinSi**

**FinSi**

**retourner** htmp1 ;

**Fin**

****

## L’arbre est-il équilibré ?

**Algorithme :** estEquilibre

**Données :**

\*a : Structure Arbre

hg, hd : entiers

**Début**

hg ← 0 ;

hd ← 0 ;

**Si** a->racine->fg != NULL **Alors**

hg ← calculHauteur(a->root->fg, 0) ;

**Finsi**

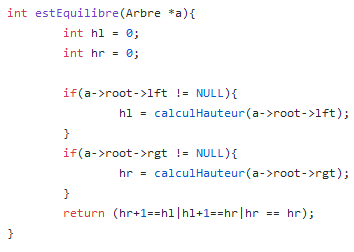
**Si** a->racine->fd != NULL **Alors**

hd ← calculHauteur(a->root->fd, 0) ;

**FinSi**

**retourner** (hd+1==hg|hg+1==hd|hg == hd) ;

**Fin**

****

## Deux Arbres sont-ils égaux ?

**Algorithme :** estEgal

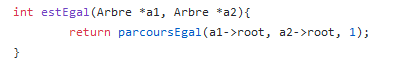
**Données :**

\*a1, \*a2 : Structure Arbre

**Début**

**retourner** parcoursEgal(a1->racine, a2->racine, 1);

**Fin**



### Parcours pour l’égalité

**Algorithme :** parcoursEgal

**Données :**

\*n1, \*n2 : structures Noeud

i, tmp : entiers

**Début**

tmp = i ;

**Si** n1 != NULL ET n2 != NULL **Alors**

**Si** n1->valeur != n2->valeur **Alors**

tmp = 0 ;

**Sinon**

tmp ← parcoursEgal(n1->fg, n2->fg, tmp);

tmp ← parcoursEgal(n1->fd, n2->fd, tmp);

**FinSi**

**Sinon**

**Si** n1 != n2 **Alors**

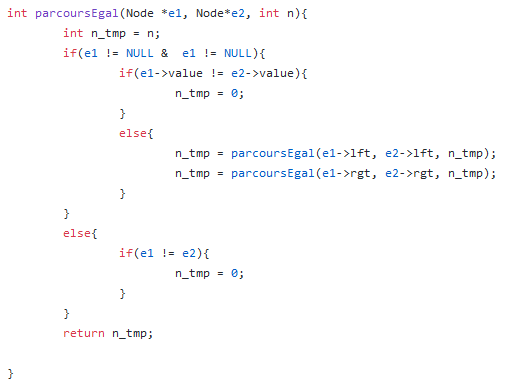
tmp ← 0;

**FinSi**

**FinSi**

**retourner** (tmp);

**Fin**



## Parcours en largeur

**Algorithme :** parcoursLargeur

**Données :**

\*p1, \*p2 : Pile de Nœuds

done : Entier

tmp : Nœud

a : Arbre

**Début**

done ← 0 ;

p1 = creerPile() ;

p2 = creerPile() ;

empilerPile(p1, a-> racine) ;

**TantQue** done != 1 **Faire**

done ← 1

//depiler et stocker nouveaux fils dans une pile

**TantQue** sommet(p1) != NULL **Faire**

empilerPile(p2, sommet(p1)->valeur->fg) ;

**Si** (sommet(p2) != NULL) **Alors**

done ← 0 ;

Ecrire(sommet(p2)->valeur->valeur) ;

**Sinon**

Ecrire(‘[]’) ;

**FinSi**

**Si**

empilerPile(p2, sommet(p1)->valeur->fd) ;

**Si** (sommet(p2) != NULL) **Alors**

done ← 0 ;

Ecrire(sommet(p2)->valeur->valeur) ;

**Sinon**

Ecrire(‘[]’) ;

**FinSi**

depilerPile(p1) ;

**FinTantQue**

//répéter tant que pile pas vide au départ (= pas de fils). Stockage des null, donc arret quand un parcours de pile = que des null

p1 = p2 ;

viderPile(p2) ;

**FinTantQue**

viderPile(p1);

viderPile(p2);

libérer(p1);

libérer(p2);

**Fin**

****

## Vida(n)ge de l’Arbre

**Algorithme :** estVide

**Données :**

\*a: Structure Arbre

**Début**

**Si** a != NULL ET a->racine != NULL **Alors**

vidage(a->racine);

**FinSi**

a->racine ← NULL;

a->size ← 0;

**Fin**

### Vidage (récursif)

**Algorithme :** vidange

**Données :**

\*e: Structure Noeud

**Début**

**S**i e != NULL **Alors**

vidange(e->fg);

vidange(e->fd);

e->fg ← NULL;

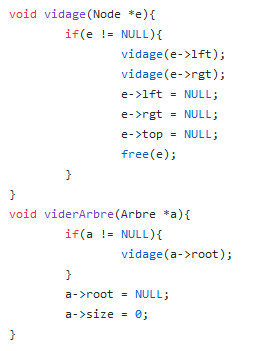
e->fd ← NULL;

e->top ← NULL;

libérer(e);

**FinSi**

**Fin**

****

## Ajout Logique

**Algorithme :** ajoutLogiqueArbre

**Données :**

\*e, \*ne : Structure Noeud

v : entier

**Début**

//On peut ajouter depuis la racine ou de n’importe quel nœud. On suppose l’Arbre dont il fait partie non-vide.

**Si** e->fg == NULL **Alors** //Notre nœud est vide à gauche

allouer(ne); //Allocation du nouvel élément

ne->valeur ← v;

ne->top ← e;

e->fg ← ne;

**Sinon**

**Si** e->fd == NULL **Alors**

allouer(ne)

ne->valeur ← v;

ne->top ← e;

e->fg ←ne;

**Sinon**

**Si** e->fg-> valeur >= e->fd->valeur **Alors**

ajoutLogiqueArbre(e->fg, v);

**Sinon**

ajoutLogiqueArbre(e->fd, v);

**FinSi**

**FinSi**

**FinSi**

**Fin**

****

## Ajout en Largeur

L’Arbre est supposé équilibré

**Algorithme :** ajoutLargeur

**Données :**

\*p1, \*p2 : Pile de Nœuds

done, v : Entier

tmp, b : Nœud

a : Arbre

**Début**

done ← 0 ;

p1 = creerPile() ;

p2 = creerPile() ;

empilerPile(p1, a-> racine) ;

allouer(b) ;

b->valeur←v ;

**TantQue** done != 1 **Faire**

done ← 1

//depiler et stocker nouveaux fils dans une pile

**TantQue** sommet(p1) != NULL **Faire**

empilerPile(p2, sommet(p1)->valeur->fg) ;

**Si** (sommet(p2) != NULL) **Alors**

done ← 0 ;

**Sinon**

sommet(p1)->valeur->fg ← b;

b->top ← sommet(p1)->valeur;

**FinSi**

**Si**

empilerPile(p2, sommet(p1)->valeur->fd) ;

**Si** (sommet(p2) != NULL) **Alors**

done ← 0 ;

**Sinon**

sommet(p1)->valeur->fd ← b;

b->top ← sommet(p1)->valeur;

**FinSi**

depilerPile(p1) ;

**FinTantQue**

//répéter tant que pile pas vide au départ (= pas de fils). Stockage des null, donc arret quand un parcours de pile = que des null

p1 ← p2 ;

viderPile(p2) ;

**FinTantQue**

viderPile(p1);

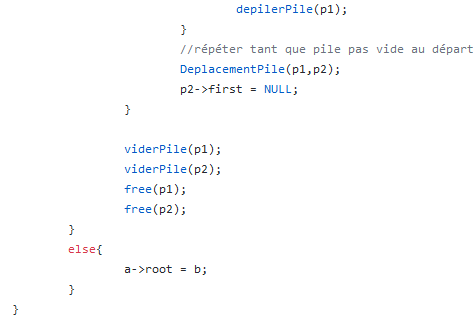
viderPile(p2);

libérer(p1);

libérer(p2) ;

**Fin**

****

****

## Equilibrage de l’arbre

**Algorithme :** equilibrageArbre

**Données :** \*a : Structure Arbre

\*p1, \*p2, \*p3 : Structure Pile

\*tmp : structure Nœud

done : Booléen

**Début**

p1 = creerPile();

p2 = creerPile();

p3 = creerPile();

done ← Faux.

empilerPile(p3, a->racine) ;

//prendre racine

empilerPile(p1, a->racine);

**TantQue** done == Faux **Faire**

done = Vrai

//depiler et stocker nouveaux fils dans une pile

**TantQue** sommet(p1) != NULL **Faire**

empilerPile(p2, sommet(p1)->valeur->fg) ;

tmp = sommet(p1)->valeur->fg ;

**Si** tmp != NULL **Alors**

done ← Faux;

empilerPile(p3, sommet(p1)->valeur->fg);

**FinSi**

empilerPile(p2, sommet(p1)->valeur->fd);

tmp ← sommet(p1)->valeur->fd;

**Si** tmp != NULL **Alors**

done ← Faux;

empilerPile(p3, sommet(p1)->valeur->fd);

**FinSi**

depilerPile(p1);

**FinTantQue**

//répéter tant que pile pas vide au départ (= pas de fils). Stockage des null, donc arret quand un parcours de pile = que des null

p1 ← p2;

viderPile(p2);

**FinTantQue**

vidage(a->racine);

a->racine ← depilerPile(p3);

**TantQue** sommet(p3) != NULL **Faire**

ajoutLargeur(a, depilerPile(p3)->valeur);

**FinTantQue**

viderPile(p1);

viderPile(p2);

viderPile(p3);

liberer(p1);

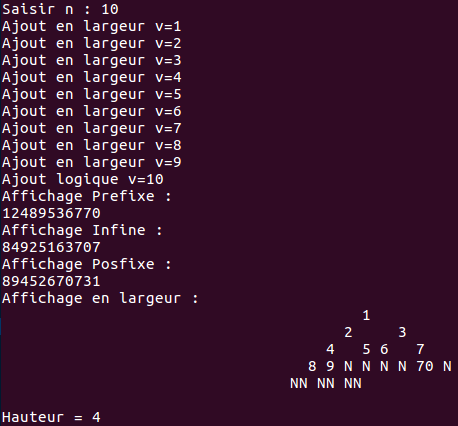
liberer(p2);

liberer (p3);

**Fin**

****

**Exemple d’Arbre :**



# Algorithmes de Huffman

Dans cette partie, nous traiterons différemment l’algorithme d’Huffman par rapport au cours (Cf TP et Mail envoyés). Nous n’utiliserons ainsi pas de foret, ce qui nous permettra d’être plus efficace et d’avoir un arbre de priorité pouvant être dynamiquement parcouru.

## Définition des structures

**Définir :** Structure données

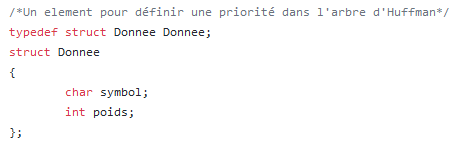
**Début**

symbole : caractère

poids : entier

**Fin**

**Nommer** Données



## Recherche du maximum

Recherche la priorité maximale dans la table de priorité. Retourne le symbole correspondant et place le symbole à la fin. Il ne sera plus visité (n est diminué de 1 au prochain appel)

**Algorithme :** rechercherMax

**Données :**

\*p: Tableau de données

tmp : Structure données

n, i, imax ; entiers

**Début**

i, imax ← 0 ;

//n est réduit n à chaque appel

**TantQue** i < n **Faire**

//On stocke l'indice où la valeur est maximale.

**Si** p[i].poids >= p[imax].poids **Alors**

imax ← i;

**FinSi**

i ← i+1

**FinTantQue**

//On échange les valeurs entre la case où la priorité est maximale et la dernière case.

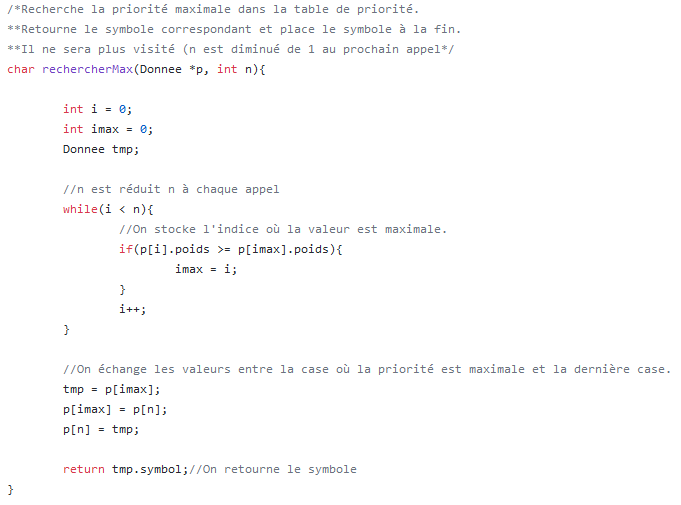
tmp ← p[imax];

p[imax] ← p[n];

p[n] ← tmp;

**retourner** tmp.symbol;//On retourne le symbole

**Fin**



## Transformation de la table de priorité

Transforme un tableau de priorité en un arbre d'Huffman. On crée l'arbre à partir du haut, donc on recherche la priorité maximale, on place son symbole sur le fils droit, puis on place notre pointeur de parcours sur le fils gauche. Si on est à la fin de la table, on place directement sur le dernier fils gauche.

**Algorithme :** transformerPrio

**Données :**

\*a: Structure Arbre

\*b, \*d : Structure Nœud d’Arbre

n, nn : Entiers

c : Caractères

**Début**

allouer(a);

allouer(a->racine);

d ← a->racine ; //On commence au sommet de l'arbre

**TantQue** nn != 0 **Faire**

c ← rechercherMax(p, nn);

nn ← nn - 1; //A chaque parcours du tableau, on réduit de 1 (la valeur à la case n a été traitée)

allouer(b);

b->valeur ← c;

d->fd ← b;

**Si** n != 0 **Alors** //On est à la fin de la création de l'arbre, on place donc le dernier symbole dans le dernier fils gauche.

allouer(d->fg);

d ← d->fg;

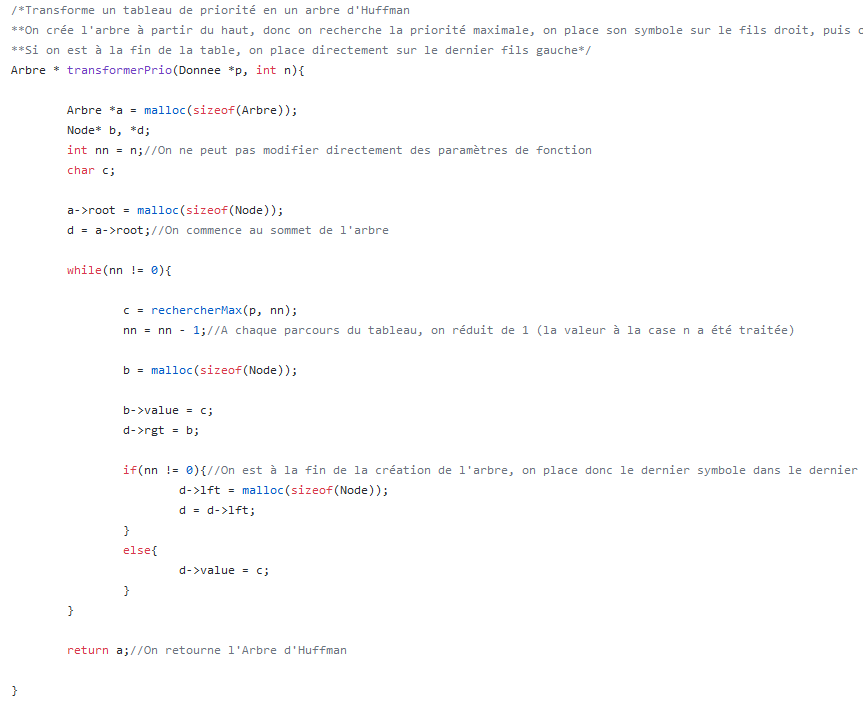
**Sinon**

d->valeur ←c;

**FinTantQue**

**retourner** a; //On retourne l'Arbre d'Huffman

**Fin**



## Décodage de Huffman

Grace à l’arbre de décodage obtenu à l’aide la table de priorité, on décode une chaine de caractères composée de 0 et de 1 (binaire). On parcourt dynamique l’arbre.

**Algorithme :** decodageHuffman

**Données :**

\*p: Table de Données

\*s : Chaine de caractères

i, n, nn : Entiers

\*a : Structure Arbre

\*b : Structure Noeud d’Arbre

**Début**

i = 0;

/\*Transformer prio retourne l'arbre d'Huffman à partir de la table de priorités\*/

\* a ← transformerPrio(p, nn);

Node \*b ← a->racine;

/\*On parcourt la chaine de caractère en caractère.\*/

**Pour** i **allant de** 0 à n **Faire**

**Si** s[i] == '1' **Alors**

Ecrire(b->rgt->value);

b ← a->racine;

**Sinon**

//Si on n'a pas 1, on passe au caractère suivant dans l'arbre de priorité d'Huffman

**Si** b->fg == NULL) **Alors**

Ecrire(b->valeur) ;

b ← a->racine ;

**Sinon**

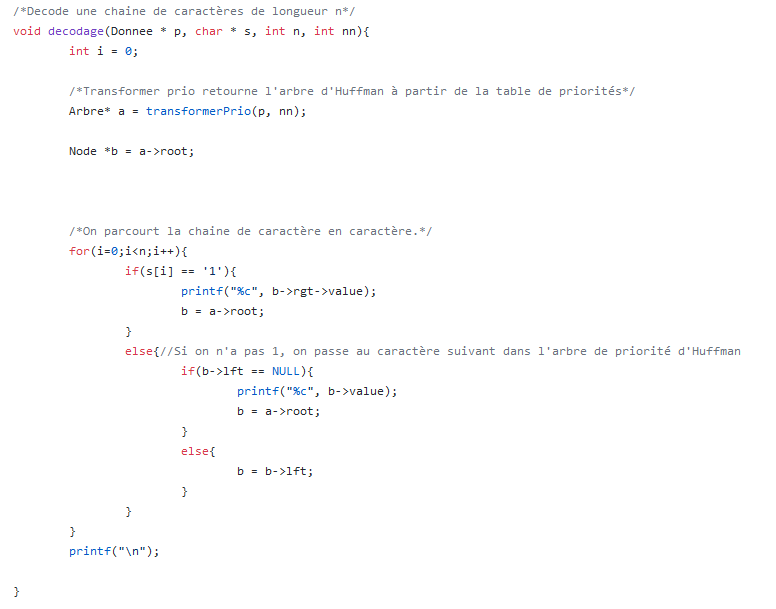
b ← b->fg;

**FinSi**

**FinSi**

**FinPour**

**Fin**



## Encodage de Huffman

Grace à l’arbre obtenu à l’aide la table de priorité, on encode une chaine de caractères en une chaine composée de 0 et de 1 (binaire). On parcourt dynamique l’arbre d’Huffman.

**Algorithme :** encodageHuffman

**Données :**

\*p: Table de Données

\*s : Chaine de caractères

i, n, nn, j : Entiers

\*a : Structure Arbre

\*b : Structure Noeud d’Arbre

**Début**

/\*Transformer prio retourner l'arbre d'Huffman à partir de la table de priorités\*/

a ← transformerPrio(p, nn);

b ← a->racine;

j ← 0;

Pour i allant de à 0 n Faire

//On regarde le fis de gauche. S'il est nul, on est arrivé tout en bas à gauche de l'arbre, donc au dernier symbole. Si on a la valeur sur le fils droit, on est arrivé au bon caractère

//j permet de compter le nombre de zéros à insérer

TantQue b->fg != NULL && b->fd->valeur != s[i]

b ← b->fg;

j ← j+1 ;

FinTantQue

Si b->fg == NULL //On n'affiche que des 0

TantQue j != 0 Faire

Ecrire(’0’) ;

j ← j – 1 ;

FinTantQue

Sinon //On affiche 1 zéro par j et le 1 une fois tous les 0 insérés

TantQue j != 0 Faire

Ecrire(‘0’) ;

j ← j -1 ;

FinTantQue

Ecrire(‘1’) ;

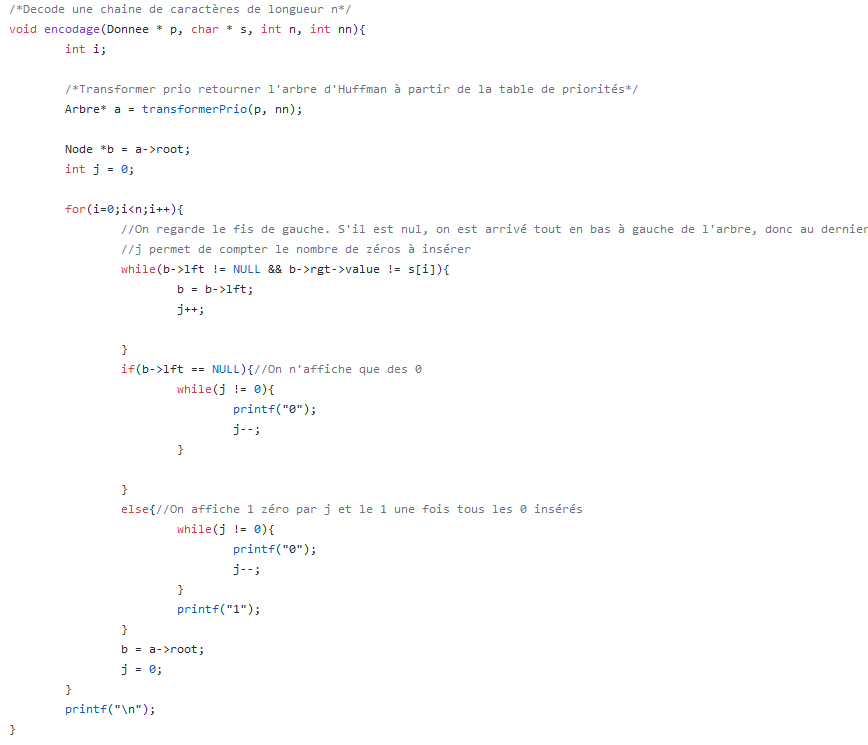
FinSi

b ← a->racine

j ← 0;

FinPour

**Fin**



# Algorithmes d’Arbre R&N

## Vérifier si un arbre R&N respecte les règles

**Algorithme :** EstRougeEtNoir

**Données :** n, hn, h : entiers;

test : booleen;

\* a : Arbre;

\* n : Nœud;

hn ← -1;

h ← 0;

n ← a → root;

**Début**

test=vrai;

// Test Rouge Et Noir

**Si** a → racine → valeur = rouge **ET** a → racine → fd = NULL **OU** a → racine → fg = NULL **OU** a → racine → fg → valeur = rouge **OU** a → racine → fd → valeur = rouge **Alors**

test=faux;

**Sinon Si** a → racine → fd = NULL **ET** a → racine → fg = NULL **Alors**

test=faux;

**Sinon**

// Test les branches Rouge Et Noir

**Debut**

**Si** n → valeur → noir **Alors**

h ← h+1;

**Si** n → fg = NULL **ET** n → fd = NULL **Alors**

**Si**  → hn = -1 **Alors**

hn ← h;

retourner vrai;

**Sinon Si** hn != h **Alors**

**FinSi**

**FinSi**

**Sinon**

EstRougeEtNoir(n->fg);

EstRougeEtNoir(n->fd);

**FinSi**

h ← h-1;

**FinSi**

**Fin**

**FinSi**

**FinSi**

retourner(test);

**Fin**